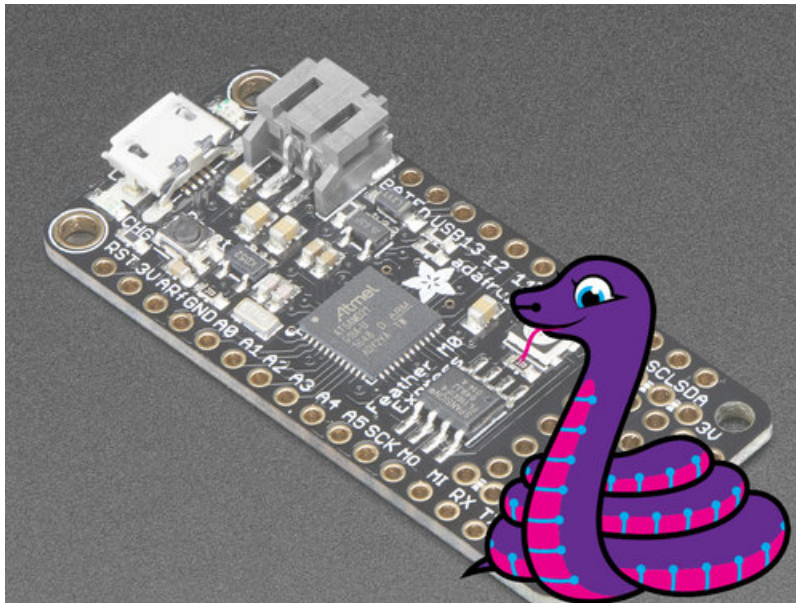


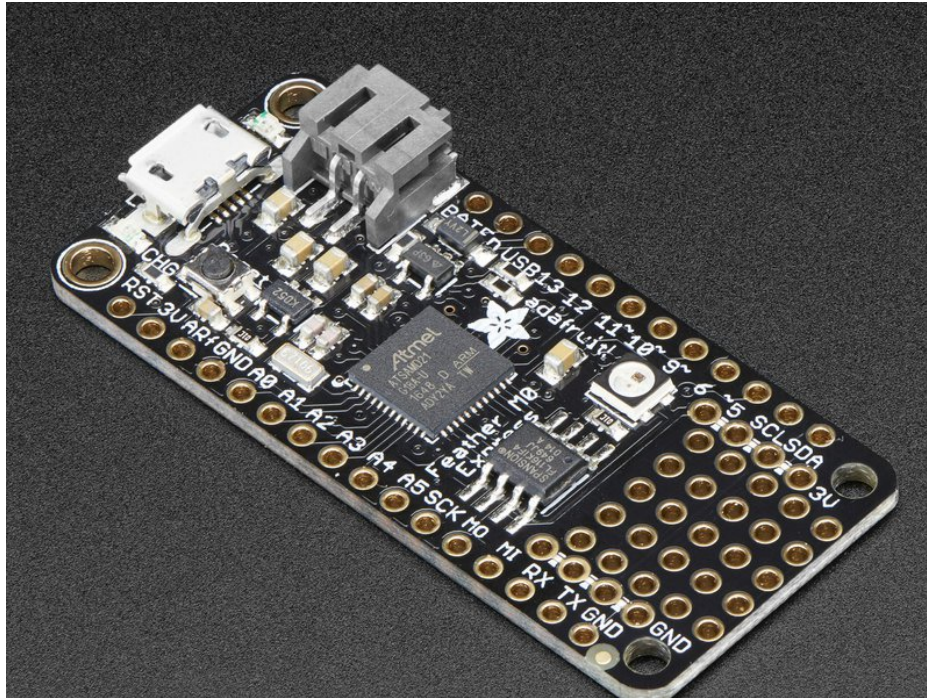
Adafruit Feather M0 Express - Designed for CircuitPython

Created by lady_ada



Last updated on 2019-07-03 08:26:44 PM UTC

Overview



We love all our Feathers equally, but this Feather is very special. It's our first Feather that is *specifically* designed for use with CircuitPython! CircuitPython is our beginner-oriented flavor of MicroPython - and as the name hints at, it's a small but full-featured version of the popular Python programming language specifically for use with circuitry and electronics.

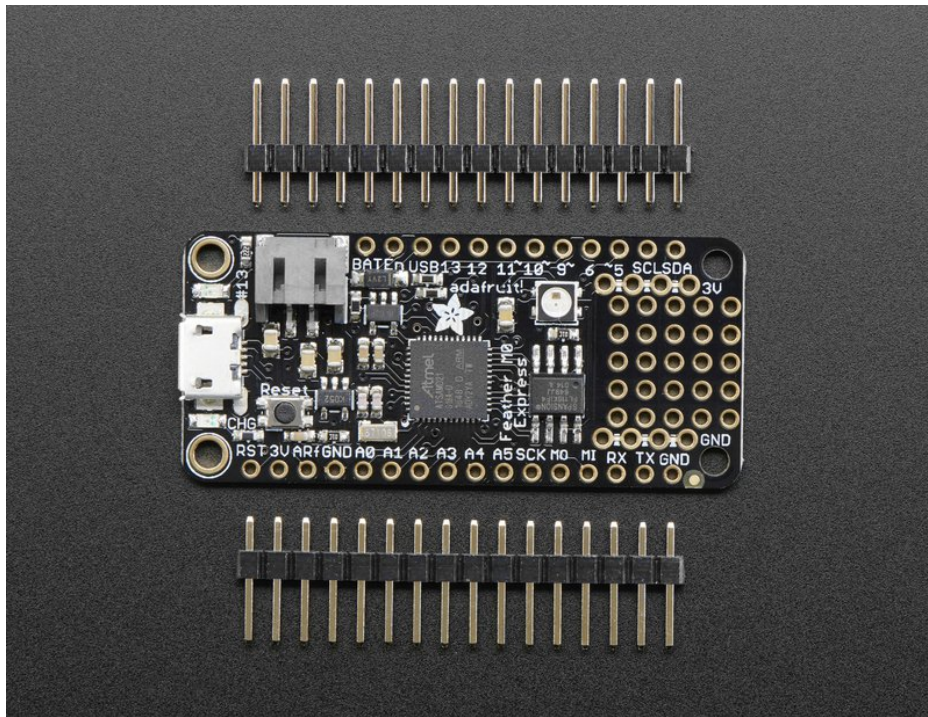


That doesn't mean you can't *also* use it with Arduino IDE! At the Feather M0's heart is an ATSAM21G18 ARM Cortex M0+ processor, clocked at 48 MHz and at 3.3V logic, the same one used in the new [Arduino Zero](http://adafru.it/2843) (<http://adafru.it/2843>). This chip has a whopping 256K of FLASH (8x more than the Atmega328 or 32u4) and 32K of RAM (16x as much)! This chip comes with built-in USB so it has USB-to-Serial program & debug capability built-in with no need for an FTDI-like chip.

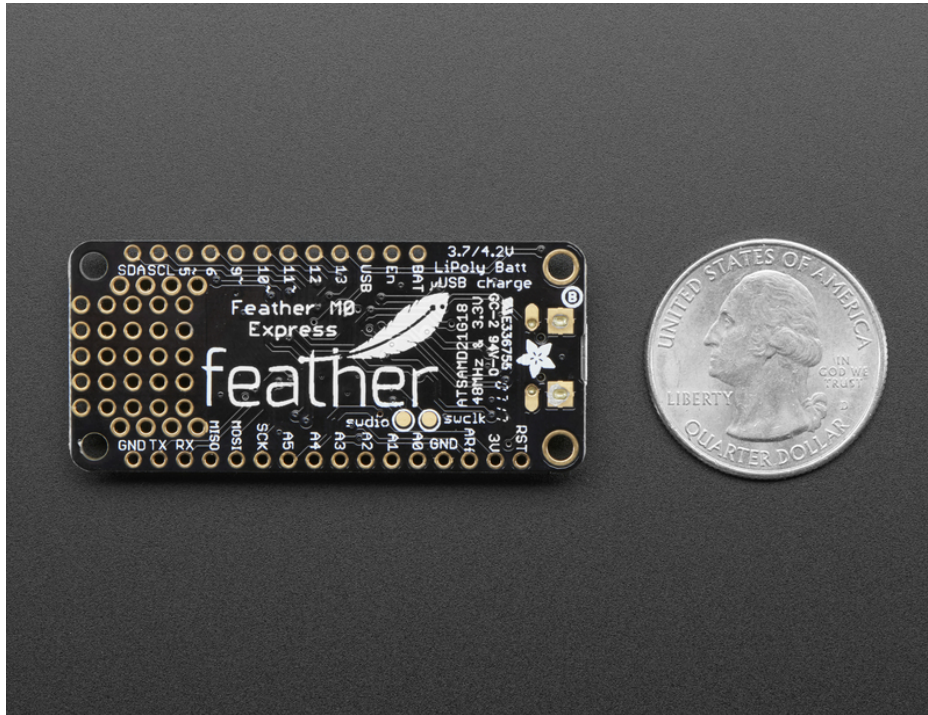
Here's some handy specs!

- Measures 2.0" x 0.9" x 0.28" (51mm x 23mm x 8mm) without headers soldered in
- Light as a (large?) feather - 5 grams

- ATSAM21G18 @ 48MHz with 3.3V logic/power
- 256KB of FLASH + 32KB of RAM
- No EEPROM
- 32.768 KHz crystal for clock generation & RTC
- 3.3V regulator with 500mA peak current output
- USB native support, comes with USB bootloader and serial port debugging
- You also get tons of pins - 20 GPIO pins
- Hardware Serial, hardware I2C, hardware SPI support
- PWM outputs on all pins
- 6 x 12-bit analog inputs
- 1 x 10-bit analog output (DAC)
- Built in 100mA lipoly charger with charging status indicator LED
- Pin #13 red LED for general purpose blinking
- Power/enable pin
- 4 mounting holes
- Reset button



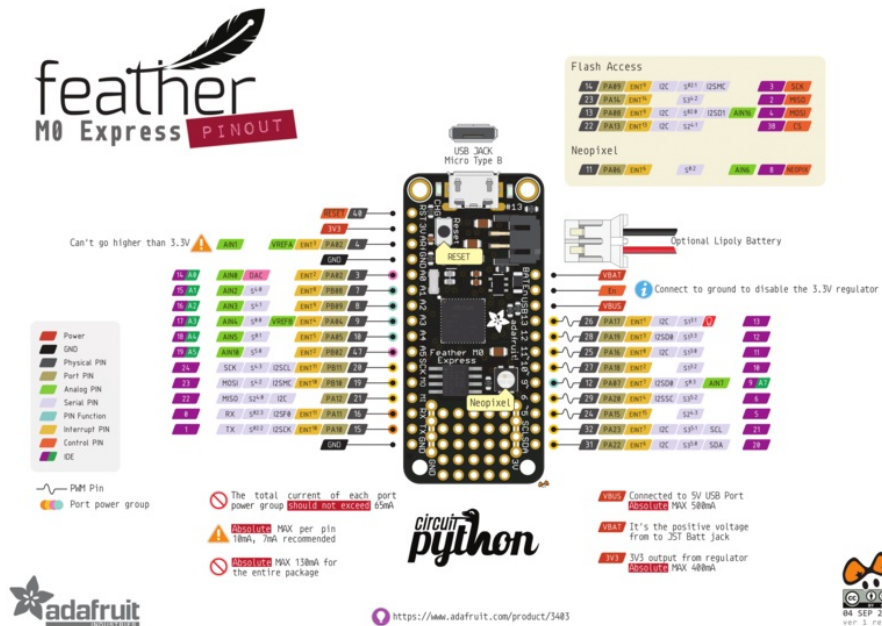
The **Feather M0 Express** uses the extra space left over to add a **Mini NeoPixel**, **2 MB SPI Flash** storage and a little prototyping space. You can use the SPI Flash storage like a very tiny hard drive. When used in Circuit Python, the 2 MB flash acts as storage for all your scripts, libraries and files. When used in Arduino, you can read/write files to it, like a little datalogger or SD card, and then with our helper program, access the files over USB.



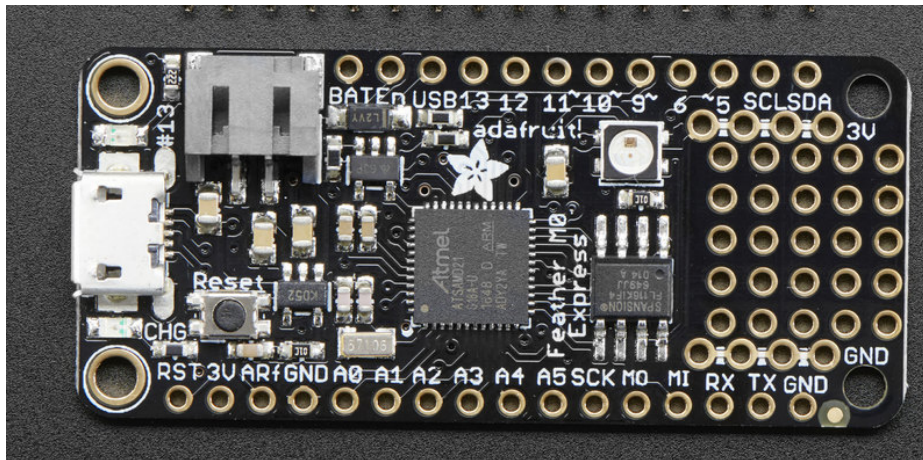
Comes fully assembled and tested, with a USB bootloader that lets you quickly use it with the Arduino IDE or for loading Circuit Python. We also toss in some header so you can solder it in and plug into a solderless breadboard.

Lipoly battery and USB cable not included (but we do have lots of options in the shop if you'd like!)

Pinouts

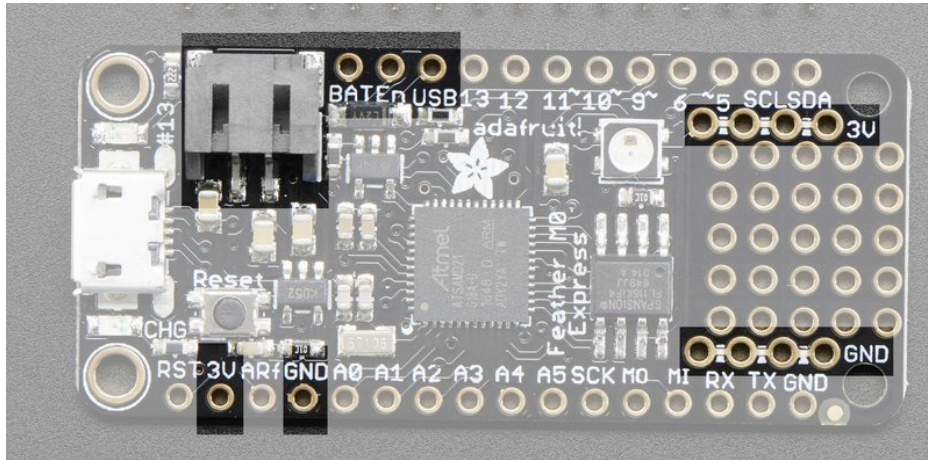


(There's a typo in the above, AREF Is PA03 not PA02)



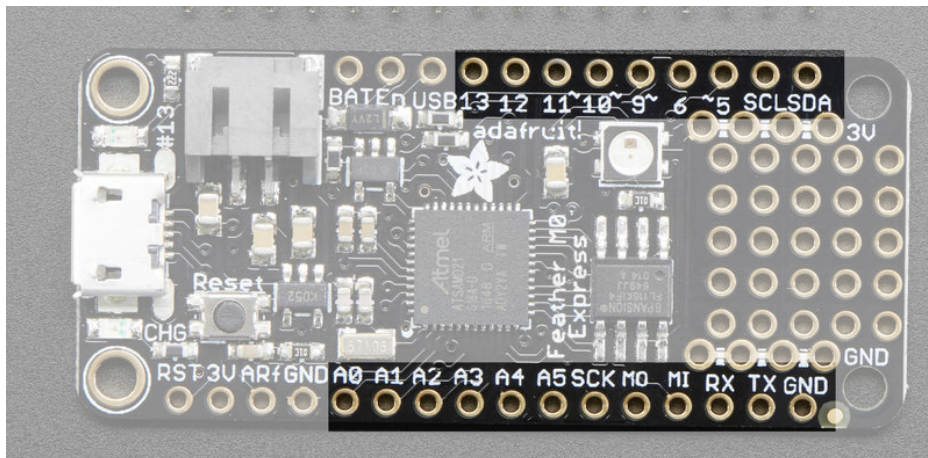
The Feather M0 is chock-full of microcontroller goodness. There's also a lot of pins and ports. We'll take you a tour of them now!

Power Pins



- **GND** - this is the common ground for all power and logic
- **BAT** - this is the positive voltage to/from the JST jack for the optional Lipoly battery
- **USB** - this is the positive voltage to/from the micro USB jack if connected
- **EN** - this is the 3.3V regulator's enable pin. It's pulled up, so connect to ground to disable the 3.3V regulator
- **3V** - this is the output from the 3.3V regulator, it can supply 500mA peak

Logic pins



This is the general purpose I/O pin set for the microcontroller.

All logic is 3.3V

Nearly all pins can do PWM output

All pins can be interrupt inputs

- **#0 / RX** - GPIO #0, also receive (input) pin for **Serial1** (hardware UART), also can be analog input
- **#1 / TX** - GPIO #1, also transmit (output) pin for **Serial1**, also can be analog input
- **SDA** - the I2C (Wire) data pin. There's no pull up on this pin by default so when using with I2C, you may need a 2.2K-10K pullup.
- **SCL** - the I2C (Wire) clock pin. There's no pull up on this pin by default so when using with I2C, you may need a 2.2K-10K pullup.
- **#5** - GPIO #5
- **#6** - GPIO #6
- **#9** - GPIO #9, also analog input **A7**. This analog input is connected to a voltage divider for the lipoly battery so be

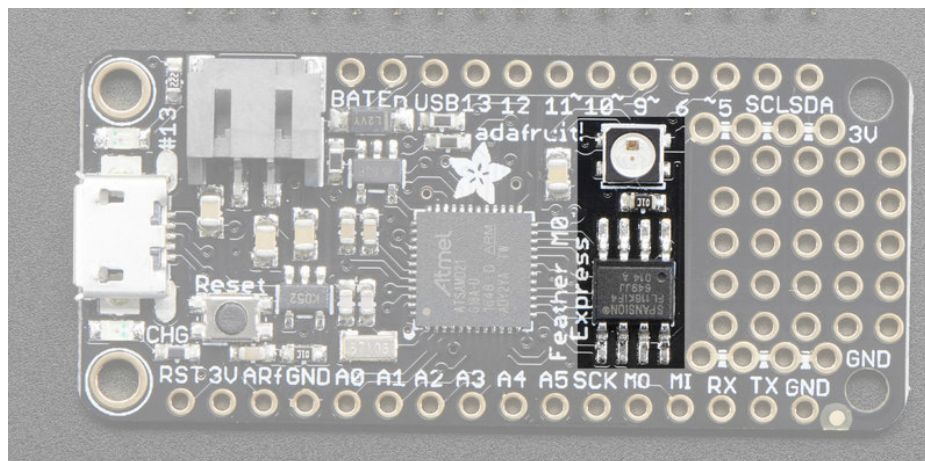
aware that this pin naturally 'sits' at around 2VDC due to the resistor divider

- **#10** - GPIO #10
- **#11** - GPIO #11
- **#12** - GPIO #12
- **#13** - GPIO #13 and is connected to the **red LED** next to the USB jack
- **A0** - This pin is analog *input* **A0** but is also an analog *output* due to having a DAC (digital-to-analog converter). You can set the raw voltage to anything from 0 to 3.3V, unlike PWM outputs this is a true analog output
- **A1 thru A5** - These are each analog input as well as digital I/O pins.
- **SCK/MOSI/MISO** - These are the hardware SPI pins, you can use them as everyday GPIO pins (but recommend keeping them free as they are best used for hardware SPI connections for high speed.)

These pins are available in CircuitPython under the `board` module. Names that start with # are prefixed with D and other names are as is. So **#0 / RX** above is available as `board.D0` and `board.RX` for example.

SPI Flash and NeoPixel

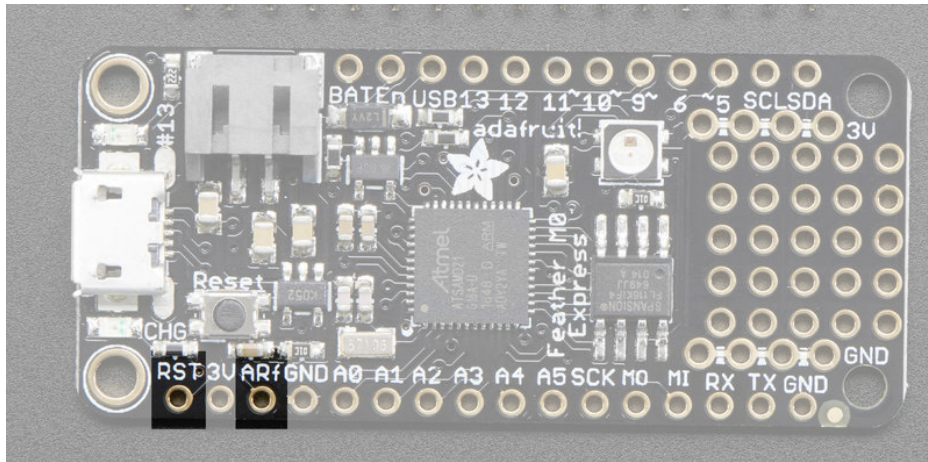
As part of the 'Express' series of boards, this Feather is designed for use with CircuitPython. To make that easy, we have added two extra parts to this Feather M0: a mini NeoPixel (RGB LED) and a 2 MB SPI Flash chip



The **NeoPixel** is connected to pin #8 in Arduino, so [just use our NeoPixel library \(https://adafru.it/dhw\)](https://adafru.it/dhw) and set it up as a single-LED strand on pin 8. The NeoPixel is powered by the 3.3V power supply but that hasn't shown to make a big difference in brightness or color. The NeoPixel is also used by the bootloader to let you know if the device has enumerated correctly (green) or USB failure (red). In CircuitPython, the LED is used to indicate the runtime status.

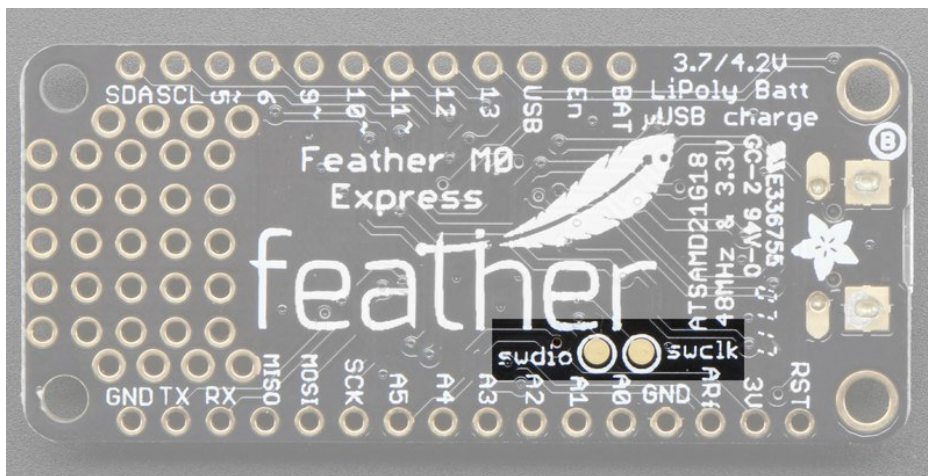
The SPI Flash is connected to 4 pins that are not brought out on the GPIO pads. This way you don't have to worry about the SPI flash colliding with other devices on the main SPI connection. Under Arduino, the FLASH **SCK** pin is #3, **MISO** is #2, **MOSI** is #4, and **CS** is #38. If you use **Feather M0 Express** as your board type, you'll be able to access the Flash SPI port under **SPI1** - this is a fully new hardware SPI device separate from the GPIO pins on the outside edge of the Feather. In CircuitPython, the SPI flash is used natively by the interpreter and is read-only to user code, instead the Flash just shows up as the writeable disk drive!

Other Pins!



- **RST** - this is the Reset pin, tie to ground to manually reset the AVR, as well as launch the bootloader manually
- **AREf** - the analog reference pin. Normally the reference voltage is the same as the chip logic voltage (3.3V) but if you need an alternative analog reference, connect it to this pin and select the external AREF in your firmware. Can't go higher than 3.3V!

Debug Interface



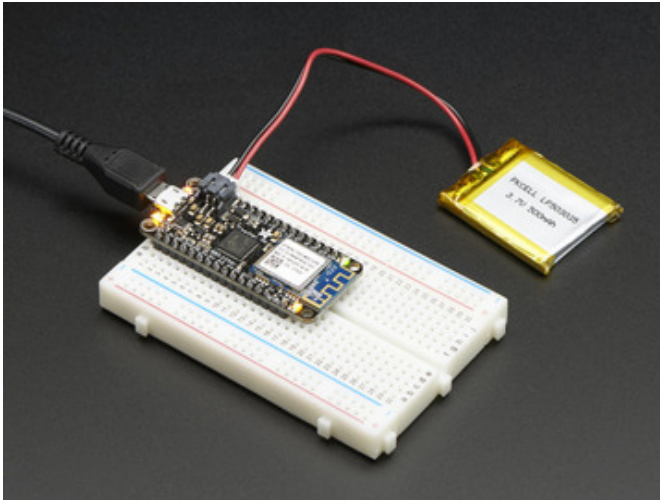
- **SWCLK & SWDIO** - These pads on the bottom are used to program the chip. They can also be connected to an SWD debugger.

Assembly

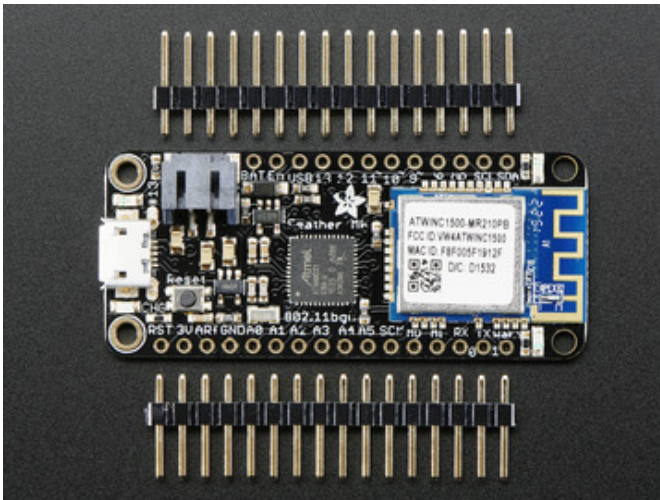
We ship Feathers fully tested but without headers attached - this gives you the most flexibility on choosing how to use and configure your Feather

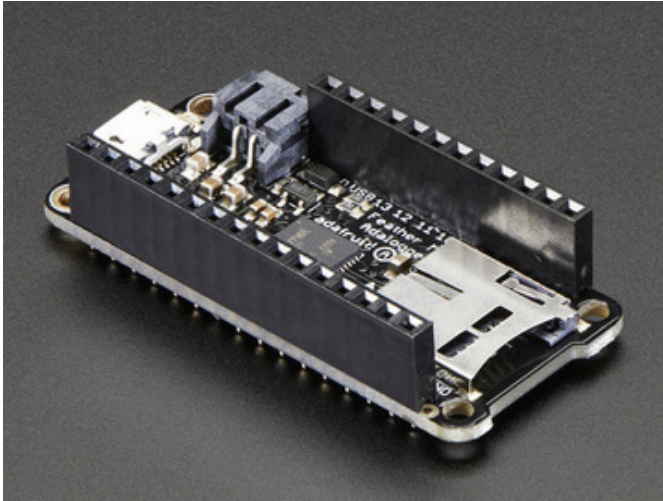
Header Options!

Before you go gung-ho on soldering, there's a few options to consider!

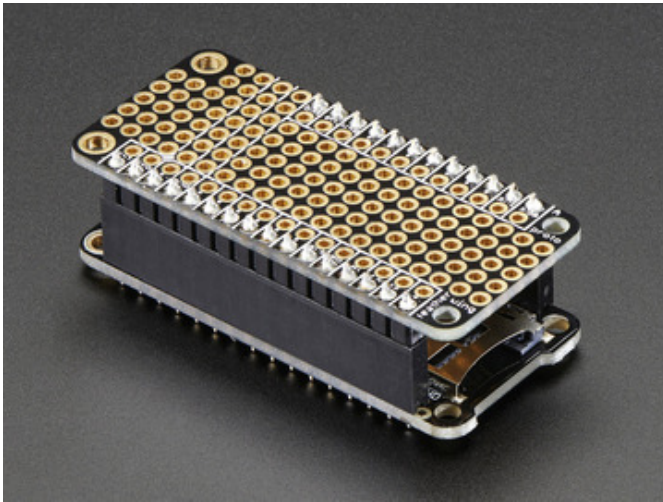


The first option is soldering in plain male headers, this lets you plug in the Feather into a solderless breadboard

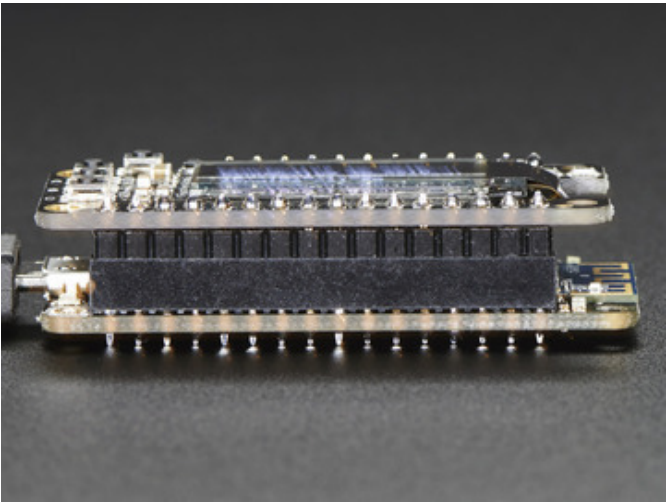
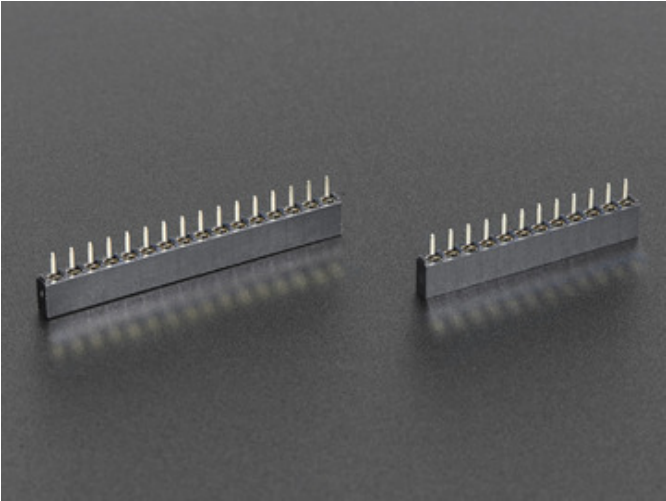


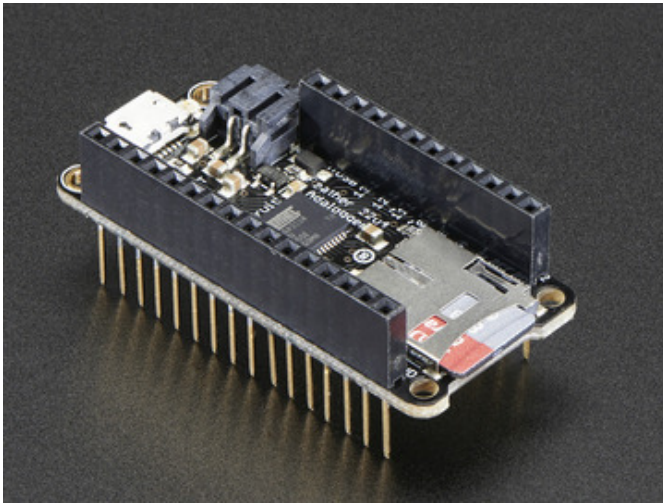


Another option is to go with socket female headers. This won't let you plug the Feather into a breadboard but it will let you attach featherwings very easily

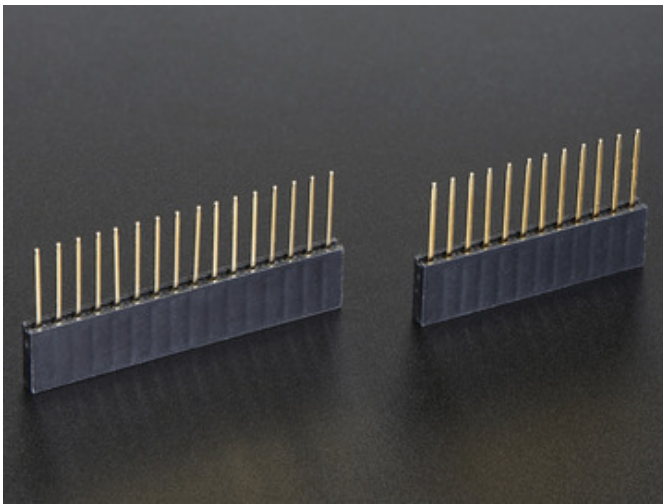


We also have 'slim' versions of the female headers, that are a little shorter and give a more compact shape

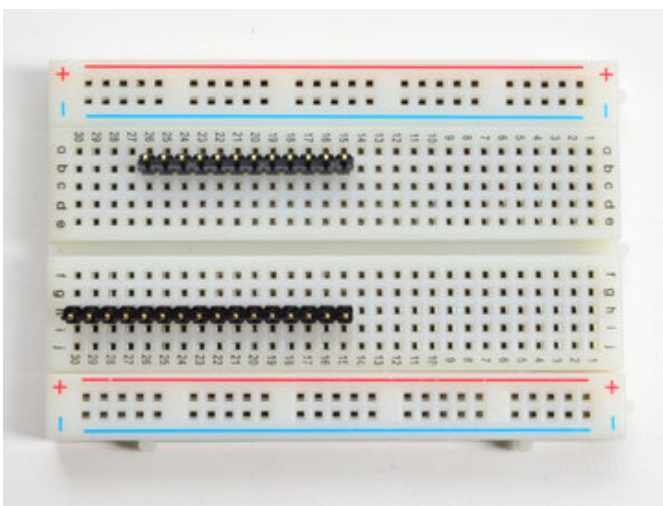




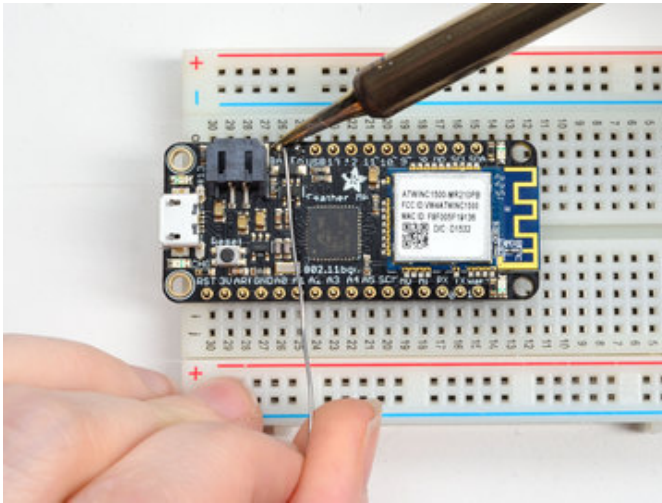
Finally, there's the "Stacking Header" option. This one is sort of the best-of-both-worlds. You get the ability to plug into a solderless breadboard *and* plug a featherwing on top. But its a little bulky



Soldering in Plain Headers



Prepare the header strip:
Cut the strip to length if necessary. It will be easier to solder if you insert it into a breadboard - **long pins down**



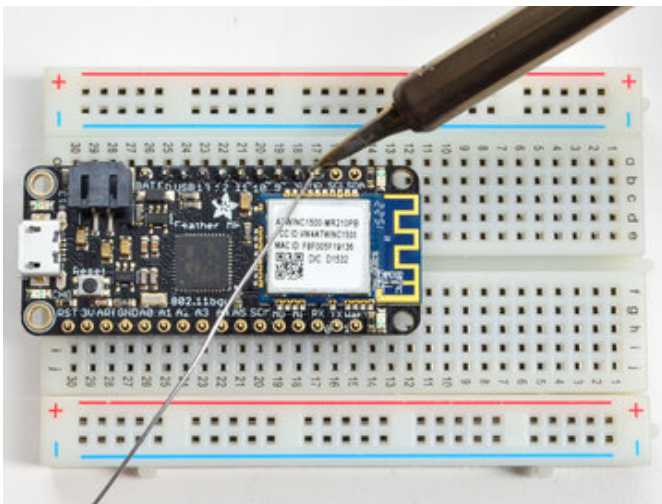
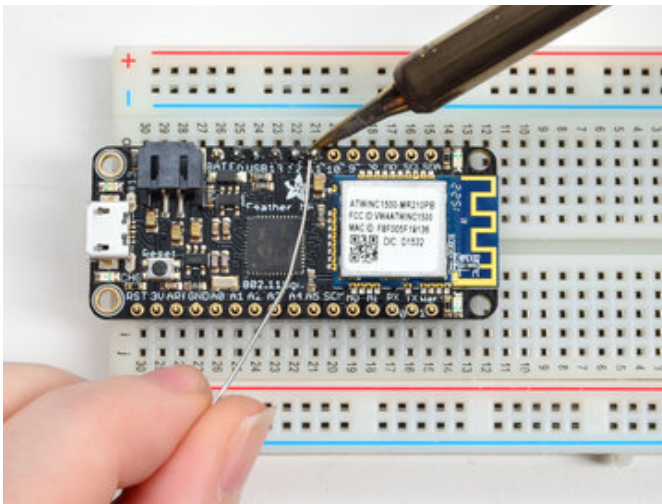
Add the breakout board:

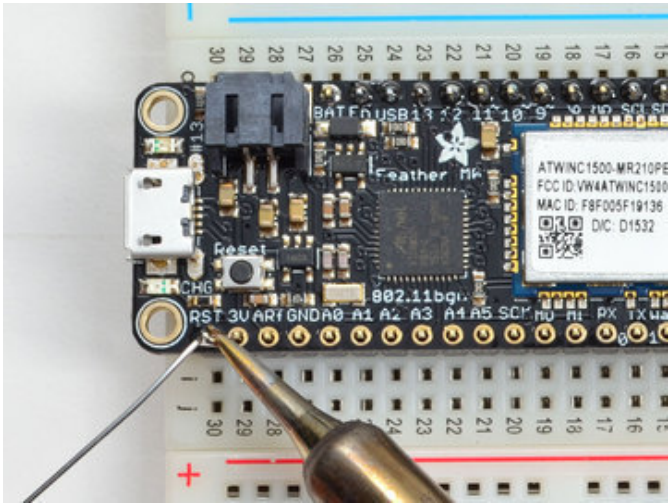
Place the breakout board over the pins so that the short pins poke through the breakout pads

And Solder!

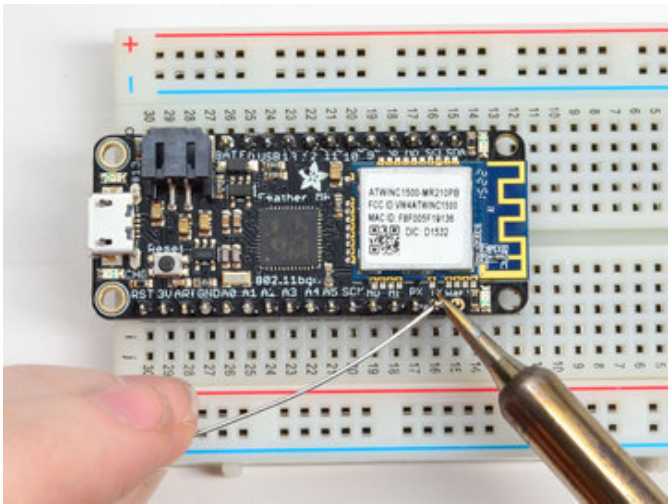
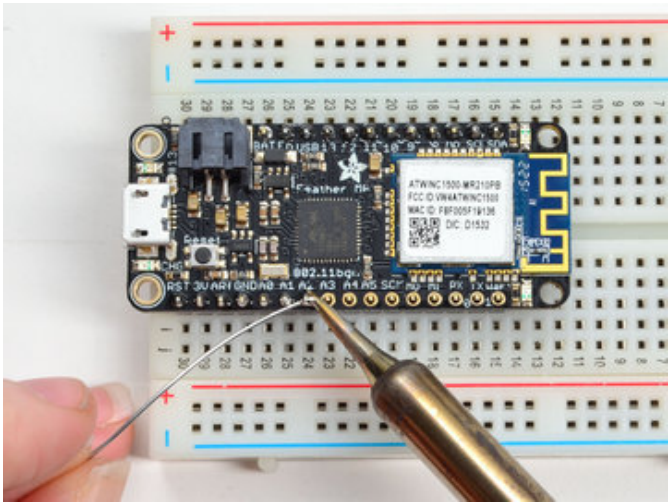
Be sure to solder all pins for reliable electrical contact.

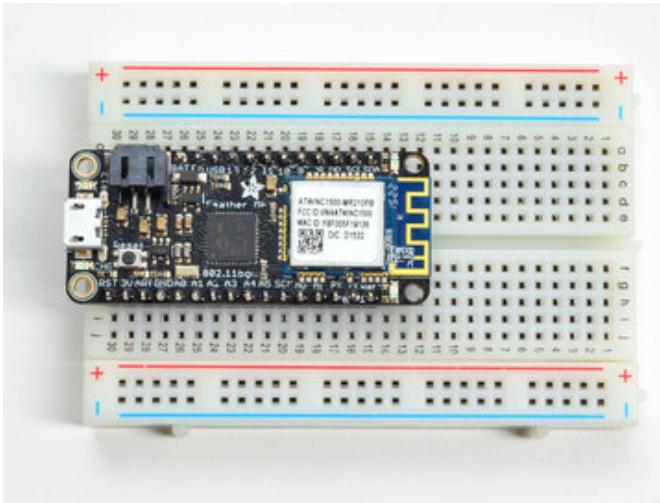
(For tips on soldering, be sure to check out our [Guide to Excellent Soldering](https://adafruit.it/aTk) (<https://adafruit.it/aTk>)).





Solder the other strip as well.





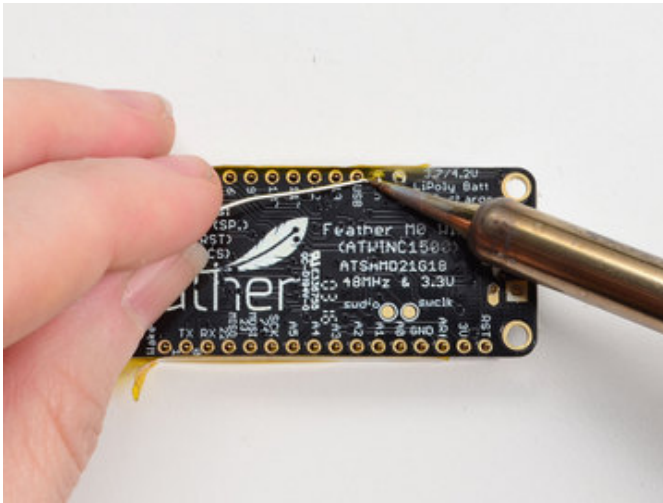
You're done! Check your solder joints visually and continue onto the next steps

Soldering on Female Header



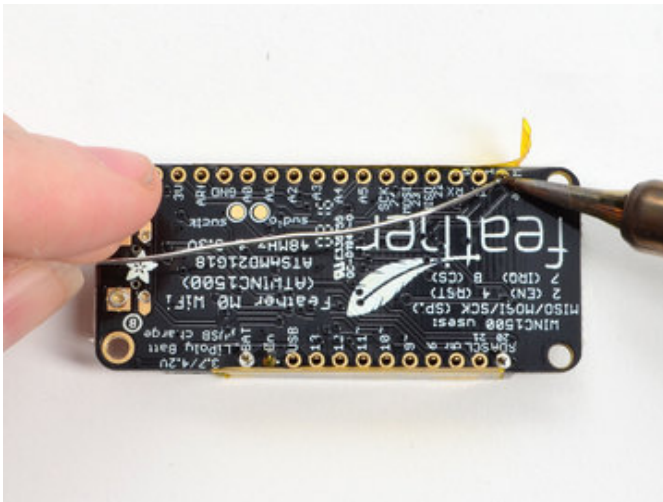
Tape In Place

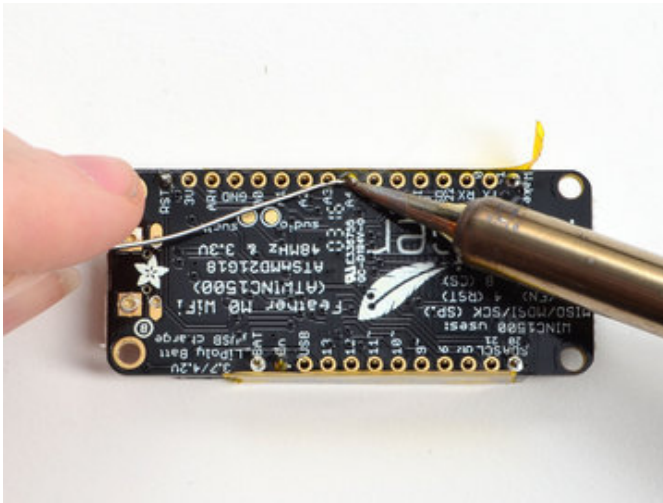
For sockets you'll want to tape them in place so when you flip over the board they don't fall out



Flip & Tack Solder

After flipping over, solder one or two points on each strip, to 'tack' the header in place

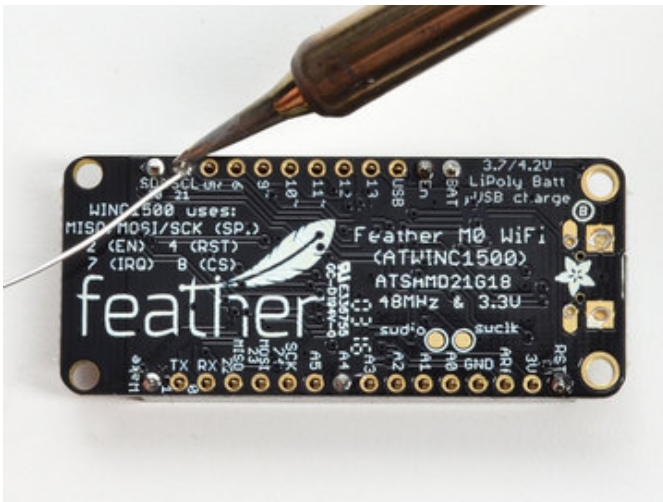


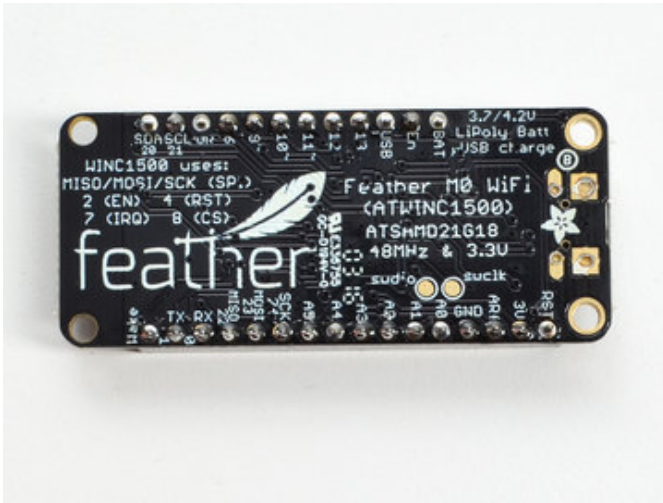


And Solder!

Be sure to solder all pins for reliable electrical contact.

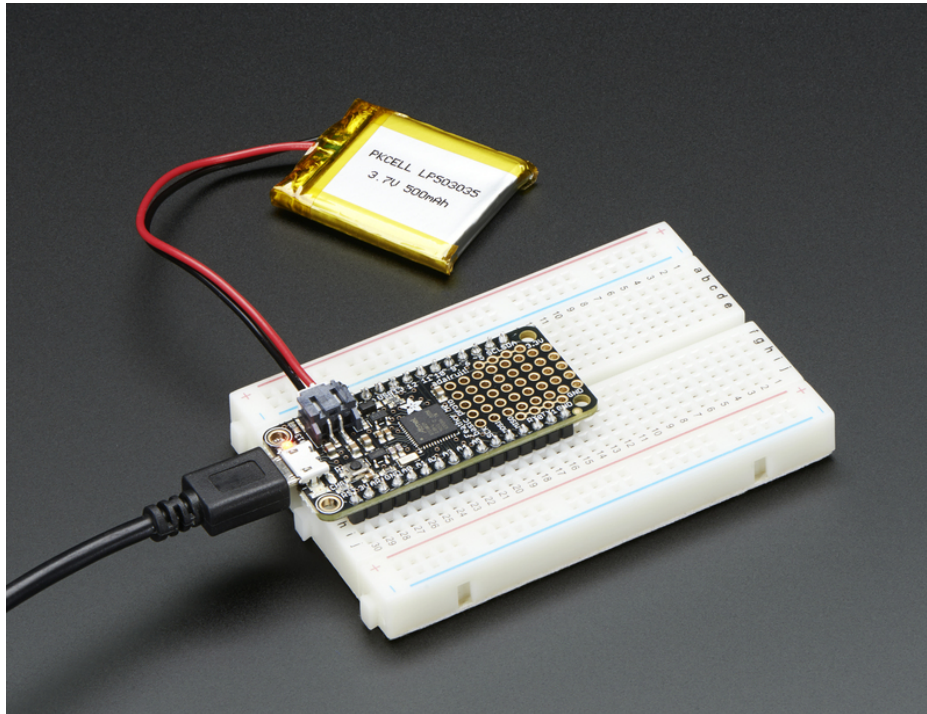
(For tips on soldering, be sure to check out our [Guide to Excellent Soldering](https://adafruit.it/aTk) (<https://adafruit.it/aTk>).





You're done! Check your solder joints visually and continue onto the next steps



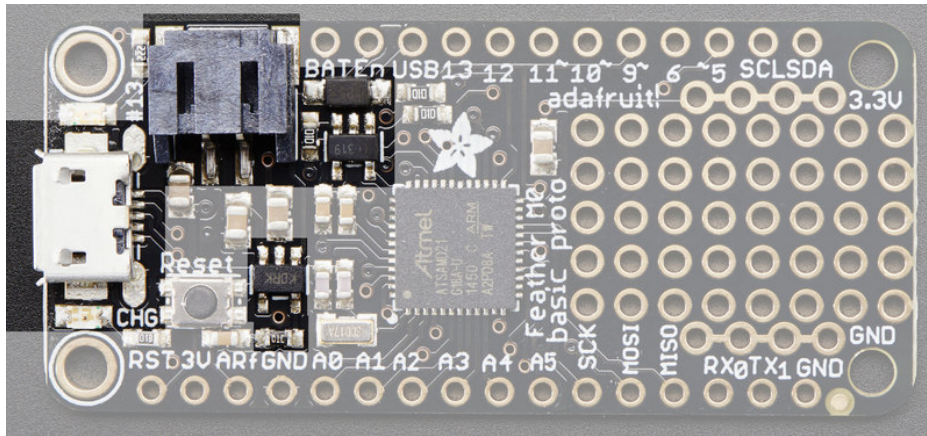


Battery + USB Power

We wanted to make the Feather easy to power both when connected to a computer as well as via battery. There's **two ways to power** a Feather. You can connect with a MicroUSB cable (just plug into the jack) and the Feather will regulate the 5V USB down to 3.3V. You can also connect a 4.2/3.7V Lithium Polymer (Lipo/Lipoly) or Lithium Ion (Lilon) battery to the JST jack. This will let the Feather run on a rechargeable battery. **When the USB power is powered, it will automatically switch over to USB for power, as well as start charging the battery (if attached) at 100mA.** This happens 'hotswap' style so you can always keep the Lipoly connected as a 'backup' power that will only get used when USB power is lost.



The JST connector polarity is matched to Adafruit LiPoly batteries. Using wrong polarity batteries can destroy your Feather



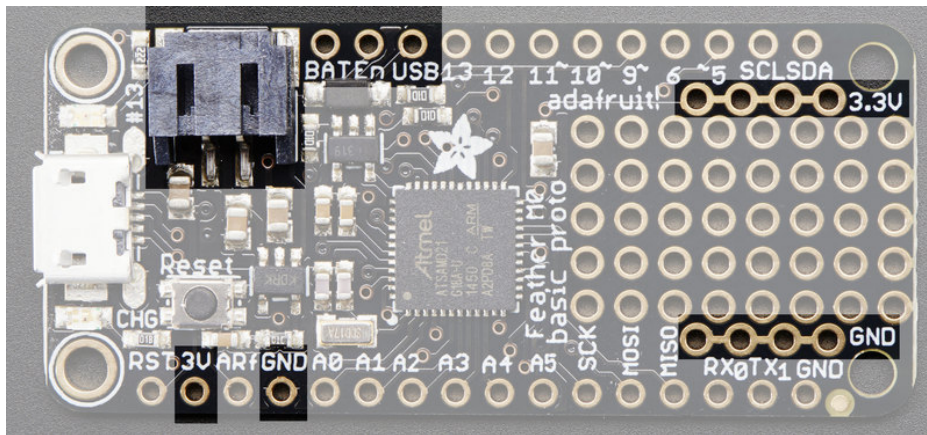
The above shows the Micro USB jack (left), Lipoly JST jack (top left), as well as the 3.3V regulator and changeover diode (just to the right of the JST jack) and the Lipoly charging circuitry (to the right of the Reset button). There's also a **CHG** LED, which will light up while the battery is charging. This LED might also flicker if the battery is not connected.



The charge LED is automatically driven by the Lipoly charger circuit. It will try to detect a battery and is expecting one to be attached. If there isn't one it may flicker once in a while when you use power because it's trying to charge a (non-existent) battery. It's not harmful, and it's totally normal!

Power supplies

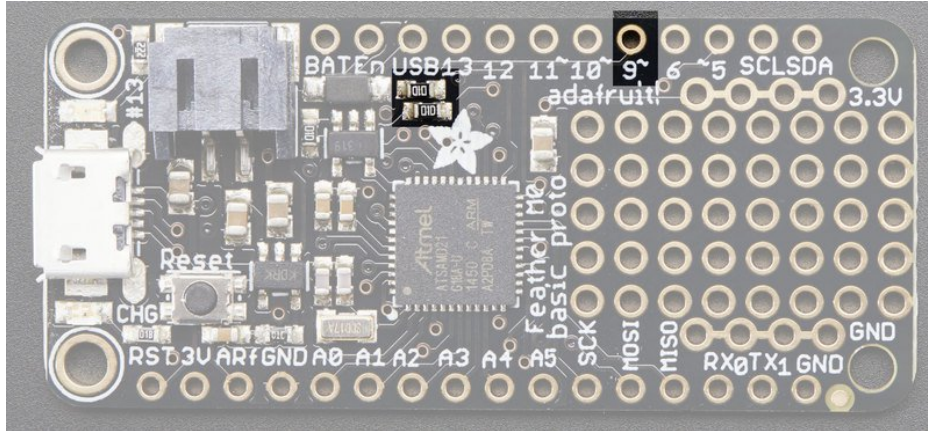
You have a lot of power supply options here! We bring out the **BAT** pin, which is tied to the lipoly JST connector, as well as **USB** which is the +5V from USB if connected. We also have the **3V** pin which has the output from the 3.3V regulator. We use a 500mA peak regulator. While you can get 500mA from it, you can't do it continuously from 5V as it will overheat the regulator. It's fine for, say, powering an ESP8266 WiFi chip or XBee radio though, since the current draw is 'spiky' & sporadic.



Measuring Battery

If you're running off of a battery, chances are you want to know what the voltage is at! That way you can tell when the battery needs recharging. Lipoly batteries are 'maxed out' at 4.2V and stick around 3.7V for much of the battery life, then slowly sink down to 3.2V or so before the protection circuitry cuts it off. By measuring the voltage you can quickly tell when you're heading below 3.7V

To make this easy we stuck a double-100K resistor divider on the **BAT** pin, and connected it to **D9** (a.k.a analog #7 **A7**).



In Arduino, you can read this pin's voltage, then double it, to get the battery voltage.

```
// Arduino Example Code

#define VBATPIN A7

float measuredvbat = analogRead(VBATPIN);
measuredvbat *= 2; // we divided by 2, so multiply back
measuredvbat *= 3.3; // Multiply by 3.3V, our reference voltage
measuredvbat /= 1024; // convert to voltage
Serial.print("VBat: "); Serial.println(measuredvbat);
```

For CircuitPython, we've written a `get_voltage()` helper function to do the math for you. All you have to do is call the function, provide the pin and print the results.

```
import board
import analogio

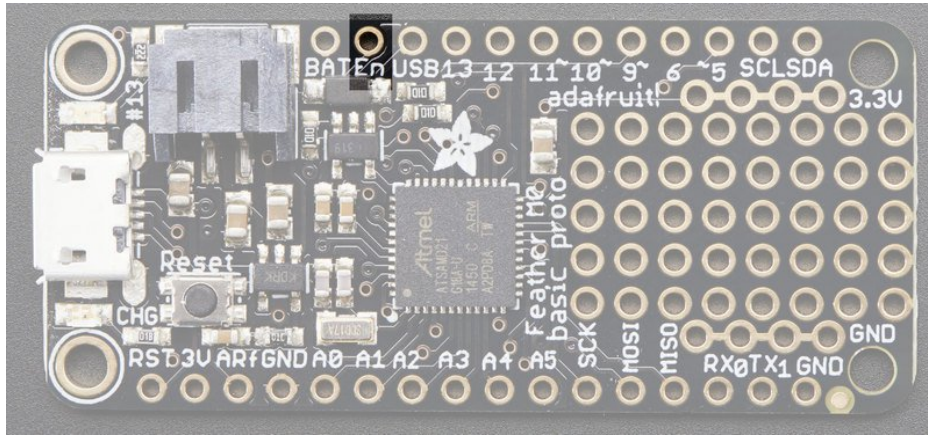
vbat_voltage = analogio.AnalogIn(board.D9)

def get_voltage(pin):
    return (pin.value * 3.3) / 65536 * 2

battery_voltage = get_voltage(vbat_voltage)
print("VBat voltage: {:.2f}".format(battery_voltage))
```

ENable pin

If you'd like to turn off the 3.3V regulator, you can do that with the **EN**(able) pin. Simply tie this pin to **Ground** and it will disable the 3V regulator. The **BAT** and **USB** pins will still be powered



Alternative Power Options

The two primary ways for powering a feather are a 3.7/4.2V LiPo battery plugged into the JST port *or* a USB power cable.

If you need other ways to power the Feather, here's what we recommend:

- For permanent installations, a [5V 1A USB wall adapter \(https://adafru.it/duP\)](https://adafru.it/duP) will let you plug in a USB cable for reliable power
- For mobile use, where you don't want a LiPoly, [use a USB battery pack! \(https://adafru.it/e2q\)](https://adafru.it/e2q)
- If you have a higher voltage power supply, [use a 5V buck converter \(https://adafru.it/DHs\)](https://adafru.it/DHs) and wire it to a [USB cable's 5V and GND input \(https://adafru.it/DHu\)](https://adafru.it/DHu)

Here's what you cannot do:

- **Do not use alkaline or NiMH batteries** and connect to the battery port - this will destroy the LiPoly charger and there's no way to disable the charger
- **Do not use 7.4V RC batteries on the battery port** - this will destroy the board

The Feather *is not designed for external power supplies* - this is a design decision to make the board compact and low cost. It is not recommended, but technically possible:

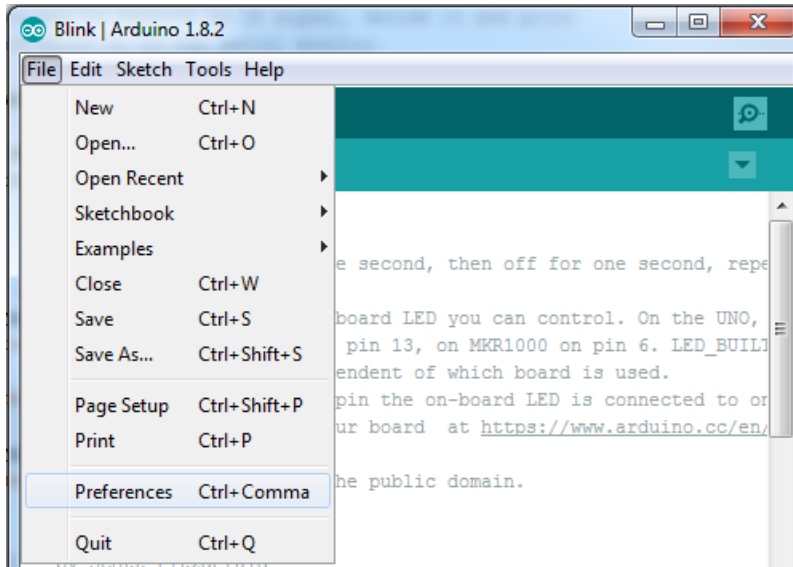
- **Connect an external 3.3V power supply to the 3V and GND pins.** Not recommended, this may cause unexpected behavior and the **EN** pin will no longer. Also this doesn't provide power on **BAT** or **USB** and some Feathers/Wings use those pins for high current usages. You may end up damaging your Feather.
- **Connect an external 5V power supply to the USB and GND pins.** Not recommended, this may cause unexpected behavior when plugging in the USB port because you will be back-powering the USB port, which *could* confuse or damage your computer.

Arduino IDE Setup

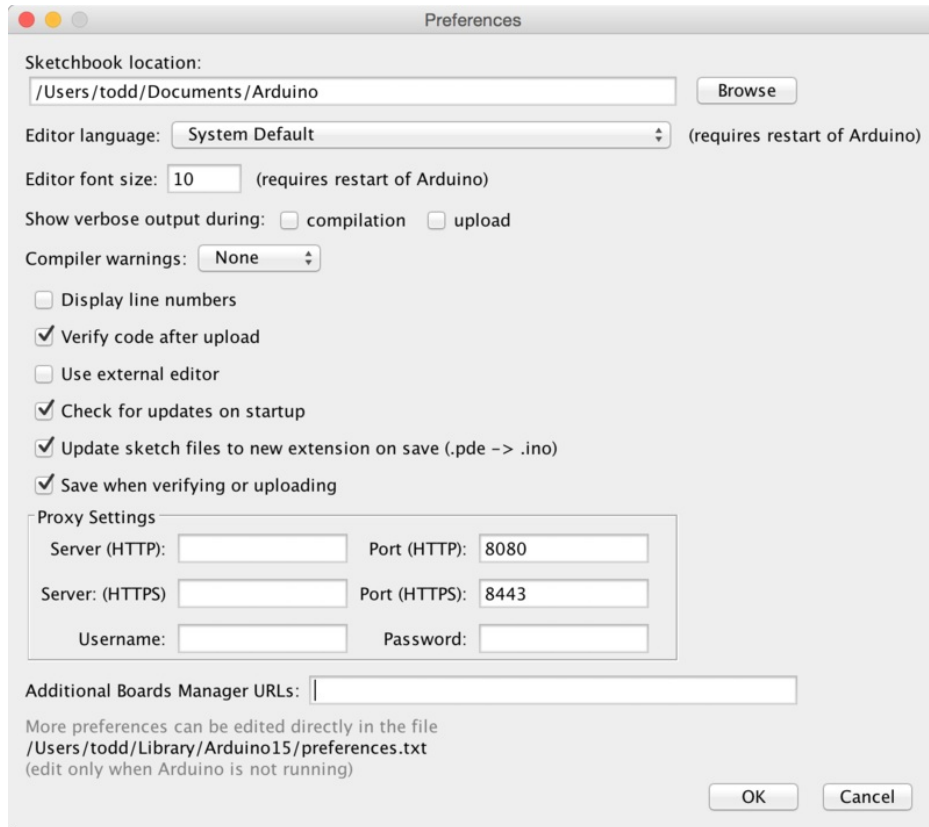
The first thing you will need to do is to download the latest release of the Arduino IDE. You will need to be using **version 1.8** or higher for this guide

<https://adafru.it/f1P>
<https://adafru.it/f1P>

After you have downloaded and installed the **latest version of Arduino IDE**, you will need to start the IDE and navigate to the **Preferences** menu. You can access it from the **File** menu in *Windows* or *Linux*, or the **Arduino** menu on *OS X*.



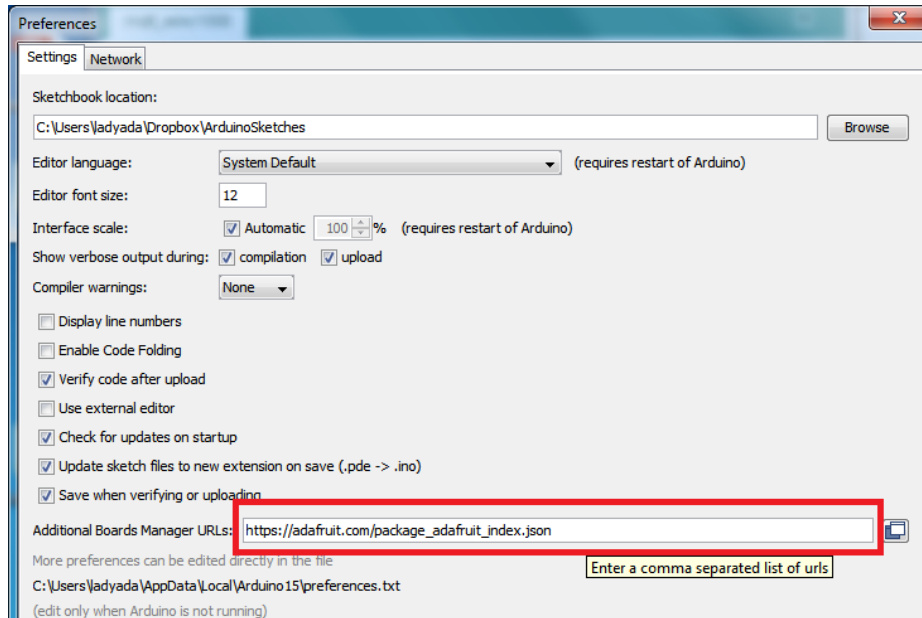
A dialog will pop up just like the one shown below.



We will be adding a URL to the new **Additional Boards Manager URLs** option. The list of URLs is comma separated, and *you will only have to add each URL once*. New Adafruit boards and updates to existing boards will automatically be picked up by the Board Manager each time it is opened. The URLs point to index files that the Board Manager uses to build the list of available & installed boards.

To find the most up to date list of URLs you can add, you can visit the list of [third party board URLs on the Arduino IDE wiki \(https://adafru.it/f7U\)](https://adafru.it/f7U). We will only need to add one URL to the IDE in this example, but *you can add multiple URLs by separating them with commas*. Copy and paste the link below into the **Additional Boards Manager URLs** option in the Arduino IDE preferences.

https://adafruit.github.io/arduino-board-index/package_adafruit_index.json



Here's a short description of each of the Adafruit supplied packages that will be available in the Board Manager when you add the URL:

- **Adafruit AVR Boards** - Includes support for Flora, Gemma, Feather 32u4, Trinket, & Trinket Pro.
- **Adafruit SAMD Boards** - Includes support for Feather M0 and M4, Metro M0 and M4, ItsyBitsy M0 and M4, Circuit Playground Express, Gemma M0 and Trinket M0
- **Arduino Leonardo & Micro MIDI-USB** - This adds MIDI over USB support for the Flora, Feather 32u4, Micro and Leonardo using the [arcore project \(https://adafru.it/eSI\)](https://adafru.it/eSI).

If you have multiple boards you want to support, say ESP8266 and Adafruit, have both URLs in the text box separated by a comma (,)

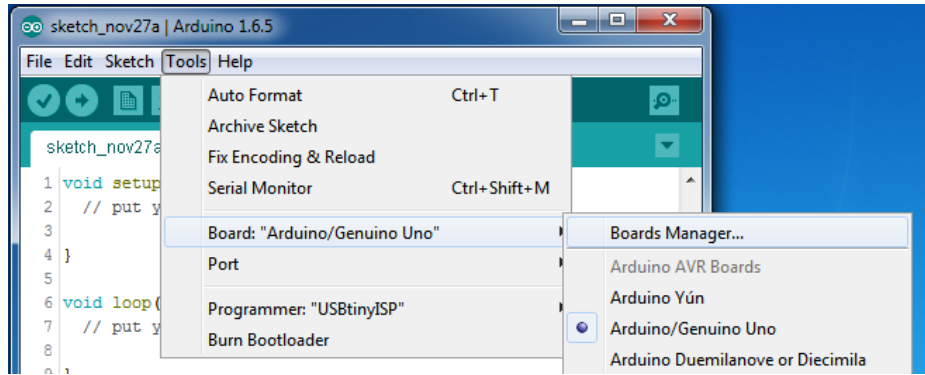
Once done click **OK** to save the new preference settings. Next we will look at installing boards with the Board Manager.

Now continue to the next step to actually install the board support package!

Using with Arduino IDE

The Feather/Metro/Gemma/Trinket M0 and M4 use an ATSAM21 or ATSAM51 chip, and you can pretty easily get it working with the Arduino IDE. Most libraries (including the popular ones like NeoPixels and display) will work with the M0 and M4, especially devices & sensors that use I2C or SPI.

Now that you have added the appropriate URLs to the Arduino IDE preferences in the previous page, you can open the **Boards Manager** by navigating to the **Tools->Board** menu.



Once the Board Manager opens, click on the category drop down menu on the top left hand side of the window and select **All**. You will then be able to select and install the boards supplied by the URLs added to the preferences.

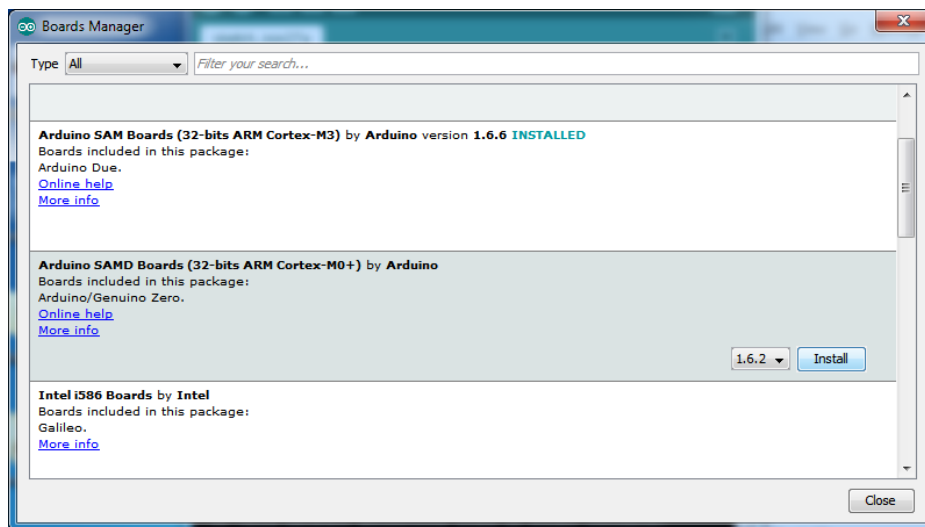


Remember you need **SETUP** the Arduino IDE to support our board packages - see the previous page on how to add adafruit's URL to the preferences

Install SAMD Support

First up, install the latest **Arduino SAMD Boards** (version **1.6.11** or later)

You can type **Arduino SAMD** in the top search bar, then when you see the entry, click **Install**

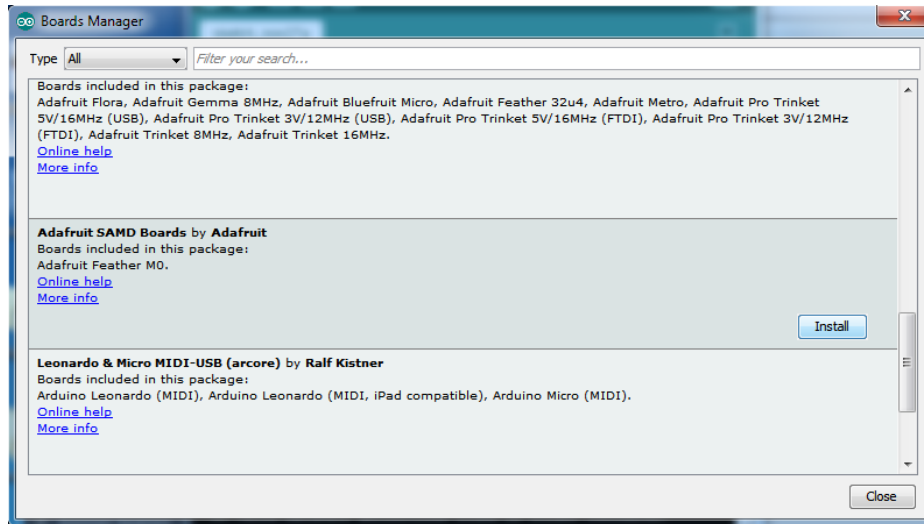


Install Adafruit SAMD

Next you can install the Adafruit SAMD package to add the board file definitions

Make sure you have **Type All** selected to the left of the *Filter your search...* box

You can type **Adafruit SAMD** in the top search bar, then when you see the entry, click **Install**

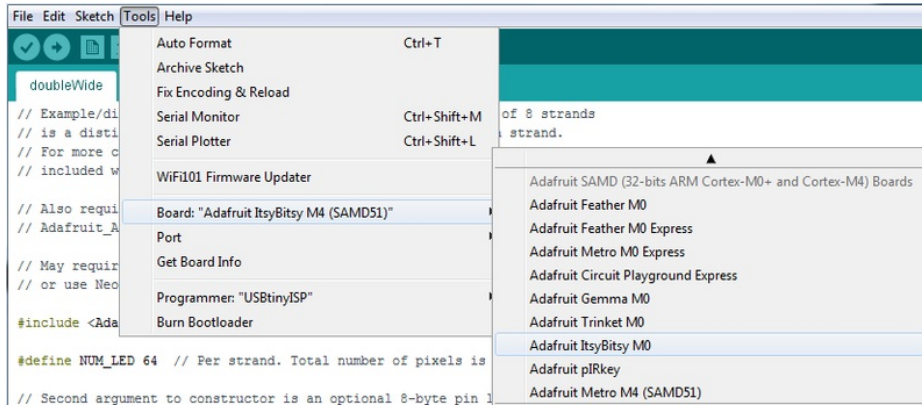


Even though in theory you don't need to - I recommend rebooting the IDE

Quit and reopen the Arduino IDE to ensure that all of the boards are properly installed. You should now be able to select and upload to the new boards listed in the **Tools->Board** menu.

Select the matching board, the current options are:

- **Feather M0** (for use with any Feather M0 other than the Express)
- **Feather M0 Express**
- **Metro M0 Express**
- **Circuit Playground Express**
- **Gemma M0**
- **Trinket M0**
- **ItsyBitsy M0**
- **Hallowing M0**
- **Crickit M0** (this is for direct programming of the Crickit, which is probably not what you want! For advanced hacking only)
- **Metro M4 Express**
- **ItsyBitsy M4 Express**
- **Feather M4 Express**
- **Trellis M4 Express**
- **Grand Central M4 Express**



Install Drivers (Windows 7 & 8 Only)

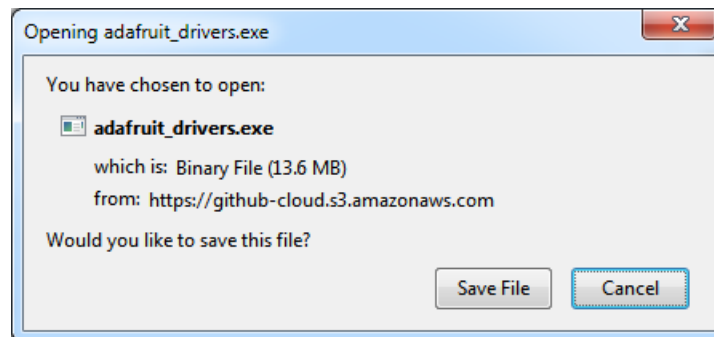
When you plug in the board, you'll need to possibly install a driver

Click below to download our Driver Installer

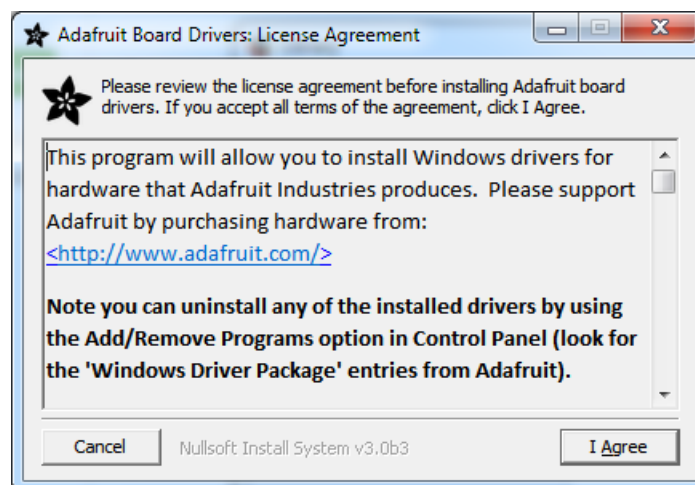
<https://adafru.it/ECO>

<https://adafru.it/ECO>

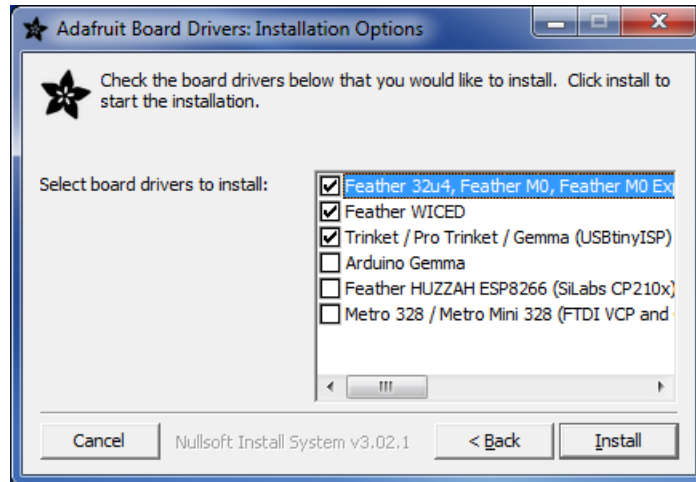
Download and run the installer



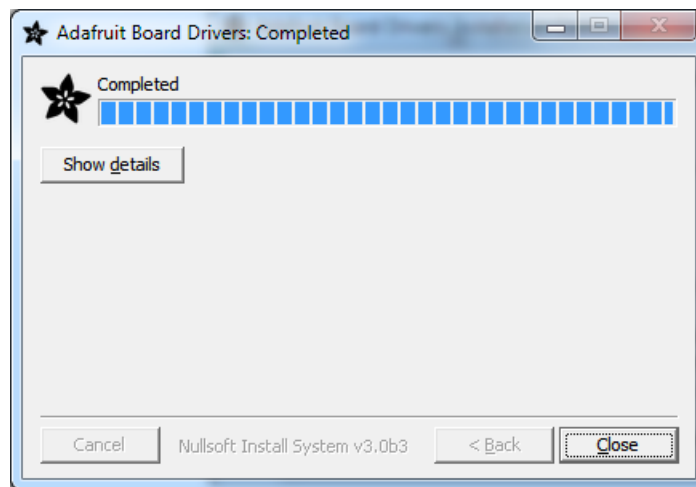
Run the installer! Since we bundle the SiLabs and FTDI drivers as well, you'll need to click through the license



Select which drivers you want to install, the defaults will set you up with just about every Adafruit board!



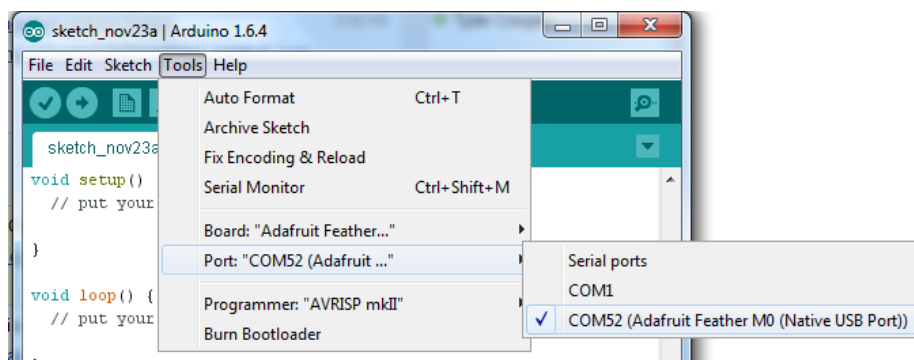
Click **Install** to do the installin'



Blink

Now you can upload your first blink sketch!

Plug in the M0 or M4 board, and wait for it to be recognized by the OS (just takes a few seconds). It will create a serial/COM port, you can now select it from the drop-down, it'll even be 'indicated' as Trinket/Gemma/Metro/Feather/ItsyBitsy/Trellis!



Now load up the Blink example

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);           // wait for a second
  digitalWrite(13, LOW); // turn the LED off by making the voltage LOW
  delay(1000);           // wait for a second
}
```

And click upload! That's it, you will be able to see the LED blink rate change as you adapt the `delay()` calls.

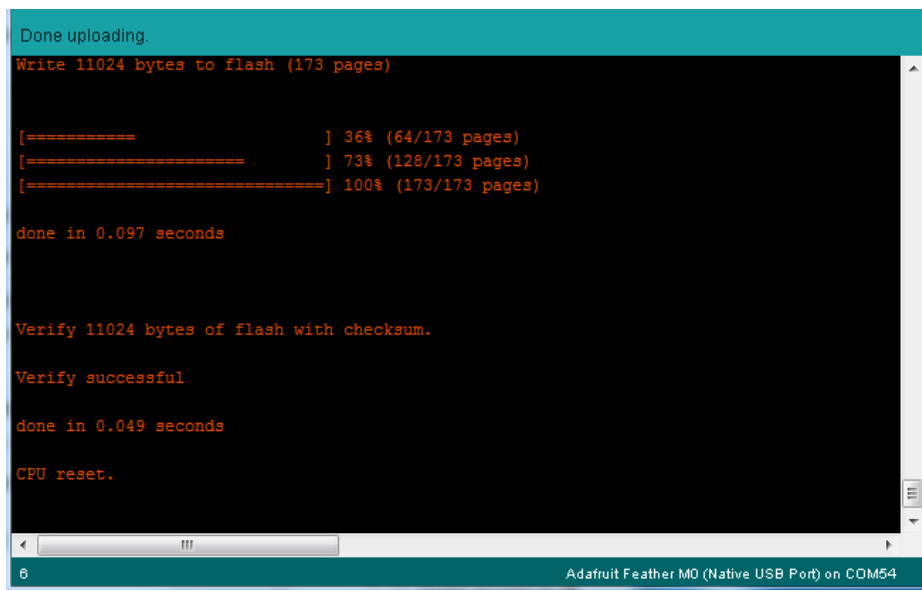
If you're using **Trellis M4 Express**, you can go to the next page cause there's no pin 13 LED - so you won't see it blink. Still this is a good thing to test compile and upload!



If you are having issues, make sure you selected the matching Board in the menu that matches the hardware you have in your hand.

Successful Upload

If you have a successful upload, you'll get a bunch of red text that tells you that the device was found and it was programmed, verified & reset



```
Done uploading.
Write 11024 bytes to flash (173 pages)

[=====] 36% (64/173 pages)
[=====] 73% (128/173 pages)
[=====] 100% (173/173 pages)

done in 0.097 seconds

Verify 11024 bytes of flash with checksum.
Verify successful

done in 0.049 seconds

CPU reset.
```

6 Adafuit Feather M0 (Native USB Port) on COM54

After uploading, you may see a message saying "Disk Not Ejected Properly" about the ...BOOT drive. You can ignore that message: it's an artifact of how the bootloader and uploading work.

The fix for this issue is to make sure Adafruit's custom udev rules are applied to your system. One of these rules is made to configure modem manager not to touch the Feather board and will fix the programming difficulty issue.

Follow the steps for installing Adafruit's udev rules on this page. (<https://adafru.it/iOE>)

Adapting Sketches to M0 & M4

The ATSAM21 and 51 are very nice little chips, but fairly new as Arduino-compatible cores go. **Most** sketches & libraries will work but here's a collection of things we noticed.

The notes below cover a range of Adafruit M0 and M4 boards, but not every rule will apply to every board (e.g. Trinket and Gemma M0 do not have ARef, so you can skip the Analog References note!).

Analog References

If you'd like to use the **ARef** pin for a non-3.3V analog reference, the code to use is `analogReference(AR_EXTERNAL)` (it's AR_EXTERNAL not EXTERNAL)

Pin Outputs & Pullups

The old-style way of turning on a pin as an input with a pullup is to use

```
pinMode(pin, INPUT)
digitalWrite(pin, HIGH)
```

This is because the pullup-selection register on 8-bit AVR chips is the same as the output-selection register.

For M0 & M4 boards, you can't do this anymore! Instead, use:

```
pinMode(pin, INPUT_PULLUP)
```

Code written this way still has the benefit of being *backwards compatible with AVR*. You don't need separate versions for the different board types.

Serial vs SerialUSB

99.9% of your existing Arduino sketches use **Serial.print** to debug and give output. For the Official Arduino SAMD/M0 core, this goes to the Serial5 port, which isn't exposed on the Feather. The USB port for the Official Arduino M0 core is called **SerialUSB** instead.

In the Adafruit M0/M4 Core, we fixed it so that **Serial goes to USB so it will automatically work just fine**.

However, on the off chance you are using the official Arduino SAMD core and *not* the Adafruit version (which really, we recommend you use our version because it's been tuned to our boards), and you want your Serial prints and reads to use the USB port, use **SerialUSB instead of **Serial** in your sketch.**

If you have existing sketches and code and you want them to work with the M0 without a huge find-replace, put

```
#if defined(ARDUINO_SAMD_ZERO) && defined(SERIAL_PORT_USBVIRTUAL)
// Required for Serial on Zero based boards
#define Serial SERIAL_PORT_USBVIRTUAL
#endif
```

right above the first function definition in your code. For example:



```
1 // Simple date conversions and calculations
2
3 #include <Wire.h>
4 #include "RTCLib.h"
5
6 #if defined(ARDUINO_ARCH_SAMD)
7 // for Zero, output on USB Serial console, remove line below if using programming port to program the Zero!
8 #define Serial SerialUSB
9 #endif
10
11 void showDate(const char* txt, const DateTime: dt) {
12     Serial.print(txt);
13     Serial.print(' ');
```

AnalogWrite / PWM on Feather/Metro M0

After looking through the SAMD21 datasheet, we've found that some of the options listed in the multiplexer table don't exist on the specific chip used in the Feather M0.

For all SAMD21 chips, there are two peripherals that can generate PWM signals: The Timer/Counter (TC) and Timer/Counter for Control Applications (TCC). Each SAMD21 has multiple copies of each, called 'instances'.

Each TC instance has one count register, one control register, and two output channels. Either channel can be enabled and disabled, and either channel can be inverted. The pins connected to a TC instance can output identical versions of the same PWM waveform, or complementary waveforms.

Each TCC instance has a single count register, but multiple compare registers and output channels. There are options for different kinds of waveform, interleaved switching, programmable dead time, and so on.

The biggest members of the SAMD21 family have five TC instances with two 'waveform output' (WO) channels, and three TCC instances with eight WO channels:

- TC[0-4],WO[0-1]
- TCC[0-2],WO[0-7]

And those are the ones shown in the datasheet's multiplexer tables.

The SAMD21G used in the Feather M0 only has three TC instances with two output channels, and three TCC instances with eight output channels:

- TC[3-5],WO[0-1]
- TCC[0-2],WO[0-7]

Tracing the signals to the pins broken out on the Feather M0, the following pins can't do PWM at all:

- **Analog pin A5**

The following pins can be configured for PWM without any signal conflicts as long as the SPI, I2C, and UART pins keep their protocol functions:

- **Digital pins 5, 6, 9, 10, 11, 12, and 13**
- **Analog pins A3 and A4**

If only the SPI pins keep their protocol functions, you can also do PWM on the following pins:

- TX and SDA (Digital pins 1 and 20)

analogWrite() PWM range

On AVR, if you set a pin's PWM with `analogWrite(pin, 255)` it will turn the pin fully HIGH. On the ARM cortex, it will set it to be 255/256 so there will be very slim but still-existing pulses-to-0V. If you need the pin to be fully on, add test code that checks if you are trying to `analogWrite(pin, 255)` and, instead, does a `digitalWrite(pin, HIGH)`

analogWrite() DAC on A0

If you are trying to use `analogWrite()` to control the DAC output on **A0**, make sure you do **not** have a line that sets the pin to output. **Remove:** `pinMode(A0, OUTPUT)`.

Missing header files

There might be code that uses libraries that are not supported by the M0 core. For example if you have a line with

```
#include <util/delay.h>
```

you'll get an error that says

```
fatal error: util/delay.h: No such file or directory
#include <util/delay.h>
      ^
compilation terminated.
Error compiling.
```

In which case you can simply locate where the line is (the error will give you the file name and line number) and 'wrap it' with `#ifdef`'s so it looks like:

```
#if !defined(ARDUINO_ARCH_SAM) && !defined(ARDUINO_ARCH_SAMD) && !defined(ESP8266) && !defined(ARDUINO_AR
#include <util/delay.h>
#endif
```

The above will also make sure that header file isn't included for other architectures

If the `#include` is in the arduino sketch itself, you can try just removing the line.

Bootloader Launching

For most other AVRs, clicking **reset** while plugged into USB will launch the bootloader manually, the bootloader will time out after a few seconds. For the M0/M4, you'll need to **double click** the button. You will see a pulsing red LED to let you know you're in bootloader mode. Once in that mode, it won't time out! Click reset again if you want to go back to launching code.

Aligned Memory Access

This is a little less likely to happen to you but it happened to me! If you're used to 8-bit platforms, you can do this nice thing where you can typecast variables around. e.g.

```
uint8_t mybuffer[4];
```

```
float f = (float)mybuffer;
```

You can't be guaranteed that this will work on a 32-bit platform because **mybuffer** might not be aligned to a 2 or 4-byte boundary. The ARM Cortex-M0 can only directly access data on 16-bit boundaries (every 2 or 4 bytes). Trying to access an odd-boundary byte (on a 1 or 3 byte location) will cause a Hard Fault and stop the MCU. Thankfully, there's an easy work around ... just use `memcpy`!

```
uint8_t mybuffer[4];  
float f;  
memcpy(&f, mybuffer, 4)
```

Floating Point Conversion

Like the AVR Arduinos, the M0 library does not have full support for converting floating point numbers to ASCII strings. Functions like `sprintf` will not convert floating point. Fortunately, the standard AVR-LIBC library includes the `dtostrf` function which can handle the conversion for you.

Unfortunately, the M0 run-time library does not have `dtostrf`. You may see some references to using `#include <avr/dtostrf.h>` to get `dtostrf` in your code. And while it will compile, it does **not** work.

Instead, check out this thread to find a working `dtostrf` function you can include in your code:

<http://forum.arduino.cc/index.php?topic=368720.0> (<https://adafru.it/IFS>)

How Much RAM Available?

The ATSAM D21G18 has 32K of RAM, but you still might need to track it for some reason. You can do so with this handy function:

```
extern "C" char *sbrk(int i);  
  
int FreeRam () {  
    char stack_dummy = 0;  
    return &stack_dummy - sbrk(0);  
}
```

Thx to <http://forum.arduino.cc/index.php?topic=365830.msg2542879#msg2542879> (<https://adafru.it/m6D>) for the tip!

Storing data in FLASH

If you're used to AVR, you've probably used **PROGMEM** to let the compiler know you'd like to put a variable or string in flash memory to save on RAM. On the ARM, it's a little easier, simply add **const** before the variable name:

```
const char str[] = "My very long string";
```

That string is now in FLASH. You can manipulate the string just like RAM data, the compiler will automatically read from FLASH so you don't need special progmem-knowledgeable functions.

You can verify where data is stored by printing out the address:

```
Serial.print("Address of str $"); Serial.println((int)&str, HEX);
```

If the address is \$2000000 or larger, it's in SRAM. If the address is between \$0000 and \$3FFFF Then it is in FLASH

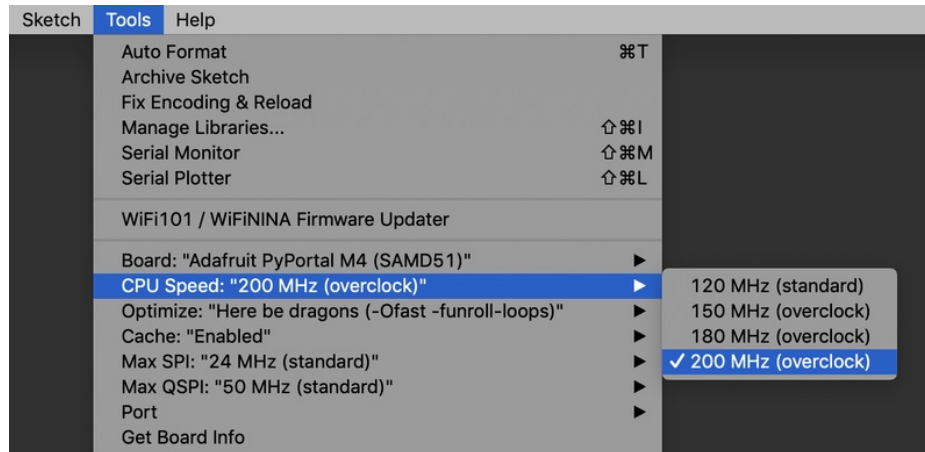
Pretty-Printing out registers

There's *a lot* of registers on the SAMD21, and you often are going through ASF or another framework to get to them. So having a way to see exactly what's going on is handy. This library from drewfish will help a ton!

<https://github.com/drewfish/arduino-ZeroRegs> (<https://adafru.it/Bet>)

M4 Performance Options

As of version 1.4.0 of the *Adafruit SAMD Boards* package in the Arduino Boards Manager, some options are available to wring extra performance out of M4-based devices. These are in the *Tools* menu.



All of these performance tweaks involve a degree of uncertainty. There's *no guarantee* of improved performance in any given project, and *some may even be detrimental*, failing to work in part or in whole. If you encounter trouble, **select the default performance settings** and re-upload.

Here's what you get and some issues you might encounter...

CPU Speed (overclocking)

This option lets you adjust the microcontroller core clock...the speed at which it processes instructions...beyond the official datasheet specifications.

Manufacturers often rate speeds conservatively because such devices are marketed for harsh industrial environments...if a system crashes, someone could lose a limb or worse. But most creative tasks are less critical and operate in more comfortable settings, and we can push things a bit if we want more speed.

There is a small but nonzero chance of code **locking up** or **failing to run** entirely. If this happens, try **dialing back the speed by one notch and re-upload**, see if it's more stable.

Much more likely, **some code or libraries may not play well** with the nonstandard CPU speed. For example, currently the NeoPixel library assumes a 120 MHz CPU speed and won't issue the correct data at other settings (this will be worked on). Other libraries may exhibit similar problems, usually anything that strictly depends on CPU timing...you might encounter problems with audio- or servo-related code depending how it's written. **If you encounter such code or libraries, set the CPU speed to the default 120 MHz and re-upload.**

Optimize

There's usually more than one way to solve a problem, some more resource-intensive than others. Since Arduino got its start on resource-limited AVR microcontrollers, the C++ compiler has always aimed for the **smallest compiled program size**. The "Optimize" menu gives some choices for the compiler to take different and often faster approaches, at the expense of slightly larger program size...with the huge flash memory capacity of M4 devices, that's rarely a problem now.

The "**Small**" setting will compile your code like it always has in the past, aiming for the smallest compiled program size.

The "**Fast**" setting invokes various speed optimizations. The resulting program should produce the same results, is slightly larger, and usually (but not always) noticeably faster. It's worth a shot!

"**Here be dragons**" invokes some more intensive optimizations...code will be larger still, faster still, but there's a possibility these optimizations could cause unexpected behaviors. *Some code may not work the same as before*. Hence the name. Maybe you'll discover treasure here, or maybe you'll sail right off the edge of the world.

Most code and libraries will continue to function regardless of the optimizer settings. If you do encounter problems, **dial it back one notch and re-upload**.

Cache

This option allows a small collection of instructions and data to be accessed more quickly than from flash memory, boosting performance. It's enabled by default and should work fine with all code and libraries. But if you encounter some esoteric situation, the cache can be disabled, then recompile and upload.

Max SPI and Max QSPI

These should probably be left at their defaults. They're present mostly for our own experiments and can cause **serious headaches**.

Max SPI determines the clock source for the M4's SPI peripherals. Under normal circumstances this allows transfers up to 24 MHz, and should usually be left at that setting. But...if you're using write-only SPI devices (such as TFT or OLED displays), this option lets you drive them faster (we've successfully used 60 MHz with some TFT screens). The caveat is, if using *any* read/write devices (such as an SD card), *this will not work at all...*SPI reads *absolutely* max out at the default 24 MHz setting, and anything else will fail. **Write = OK. Read = FAIL.** This is true *even if your code is using a lower bitrate setting...*just having the different clock source prevents SPI reads.

Max QSPI does similarly for the extra flash storage on M4 "Express" boards. *Very few* Arduino sketches access this storage at all, let alone in a bandwidth-constrained context, so this will benefit next to nobody. Additionally, due to the way clock dividers are selected, this will only provide some benefit when certain "CPU Speed" settings are active. Our [PyPortal Animated GIF Display \(https://adafru.it/EkO\)](https://adafru.it/EkO) runs marginally better with it, if using the QSPI flash.

Enabling the Buck Converter on some M4 Boards

If you want to reduce power draw, some of our boards have an inductor so you can use the 1.8V buck converter instead of the built in linear regulator. If the board does have an inductor (see the schematic) you can add the line `SUPC->VREG.bit.SEL = 1;` to your code to switch to it. Note it will make ADC/DAC reads a bit noisier so we don't use it by default. [You'll save ~4mA \(https://adafru.it/FOH\)](https://adafru.it/FOH).

Using SPI Flash

One of the best features of the M0 express board is a small SPI flash memory chip built into the board. This memory can be used for almost any purpose like storing data files, Python code, and more. Think of it like a little SD card that is always connected to the board, and in fact with Arduino you can access the memory using a library that is very similar to the [Arduino SD card library \(https://adafru.it/ucu\)](https://adafru.it/ucu). You can even read and write files that CircuitPython stores on the flash chip!

To use the flash memory with Arduino you'll need to install the [Adafruit SPI Flash Memory library \(https://adafru.it/wbt\)](https://adafru.it/wbt) in the Arduino IDE. Click the button below to download the source for this library, open the zip file, and then copy it into an **Adafruit_SPIFlash** folder (remove the -master GitHub adds to the downloaded zip and folder) [in the Arduino library folder on your computer \(https://adafru.it/dNR\)](https://adafru.it/dNR):

<https://adafru.it/wbu>

<https://adafru.it/wbu>

Once the library is installed open the Arduino IDE and look for the following examples in the library:

- `fatfs_circuitpython`
- `fatfs_datalogging`
- `fatfs_format`
- `fatfs_full_usage`
- `fatfs_print_file`
- `flash_erase`

These examples allow you to format the flash memory with a FAT filesystem (the same kind of filesystem used on SD cards) and read and write files to it just like a SD card.

Read & Write CircuitPython Files

The `fatfs_circuitpython` example shows how to read and write files on the flash chip so that they're accessible from CircuitPython. This means you can run a CircuitPython program on your board and have it store data, then run an Arduino sketch that uses this library to interact with the same data.

Note that before you use the `fatfs_circuitpython` example you **must** have loaded CircuitPython on your board. [Load the latest version of CircuitPython as explained in this guide \(https://adafru.it/BeN\)](https://adafru.it/BeN) first to ensure a CircuitPython filesystem is initialized and written to the flash chip. Once you've loaded CircuitPython then you can run the `fatfs_circuitpython` example sketch.

To run the sketch load it in the Arduino IDE and upload it to the Feather/Metro/ItsyBitsy M0 board. Then open the serial monitor at 115200 baud. You should see the serial monitor display messages as it attempts to read files and write to a file on the flash chip. Specifically the example will look for a `boot.py` and `main.py` file (like what CircuitPython runs when it starts) and print out their contents. Then it will add a line to the end of a `data.txt` file on the board (creating it if it doesn't exist already). After running the sketch you can reload CircuitPython on the board and open the `data.txt` file to read it from CircuitPython!

To understand how to read & write files that are compatible with CircuitPython let's examine the sketch code. First notice an instance of the `Adafruit_M0_Express_CircuitPython` class is created and passed an instance of the flash chip class in the last line below:

```

#define FLASH_SS      SS1                // Flash chip SS pin.
#define FLASH_SPI_PORT SPI1             // What SPI port is Flash on?

Adafruit_SPIFlash flash(FLASH_SS, &FLASH_SPI_PORT);    // Use hardware SPI

// Alternatively you can define and use non-SPI pins!
//Adafruit_SPIFlash flash(SCK1, MIS01, MOSI1, FLASH_SS);

// Finally create an Adafruit_M0_Express_CircuitPython object which gives
// an SD card-like interface to interacting with files stored in CircuitPython's
// flash filesystem.
Adafruit_M0_Express_CircuitPython pythonfs(flash);

```

By using this **Adafruit_M0_Express_CircuitPython** class you'll get a filesystem object that is compatible with reading and writing files on a CircuitPython-formatted flash chip. This is very important for interoperability between CircuitPython and Arduino as CircuitPython has specialized partitioning and flash memory layout that isn't compatible with simpler uses of the library (shown in the other examples).

Once an instance of the **Adafruit_M0_Express_CircuitPython** class is created (called **pythonfs** in this sketch) you can go on to interact with it just like if it were the [SD card library in Arduino \(https://adafru.it/wbw\)](https://adafru.it/wbw). You can open files for reading & writing, create directories, delete files and directories and more. Here's how the sketch checks if a **boot.py** file exists and prints it out a character at a time:

```

// Check if a boot.py exists and print it out.
if (pythonfs.exists("boot.py")) {
  File bootPy = pythonfs.open("boot.py", FILE_READ);
  Serial.println("Printing boot.py...");
  while (bootPy.available()) {
    char c = bootPy.read();
    Serial.print(c);
  }
  Serial.println();
}
else {
  Serial.println("No boot.py found...");
}

```

Notice the **exists** function is called to check if the **boot.py** file is found, and then the **open** function is used to open it in read mode. Once a file is opened you'll get a reference to a File class object which you can read and write from as if it were a Serial device (again just like the SD card library, [all of the same File class functions are available \(https://adafru.it/wbw\)](https://adafru.it/wbw)). In this case the **available** function will return the number of bytes left to read in the file, and the **read** function will read a character at a time to print it to the serial monitor.

Writing a file is just as easy, here's how the sketch writes to **data.txt**:


```

// Create or append to a data.txt file and add a new line
// to the end of it. CircuitPython code can later open and
// see this file too!
File data = pythonfs.open("data.txt", FILE_WRITE);
if (data) {
  // Write a new line to the file:
  data.println("Hello CircuitPython from Arduino!");
  data.close();
  // See the other fatfs examples like fatfs_full_usage and fatfs_datalogging
  // for more examples of interacting with files.
  Serial.println("Wrote a new line to the end of data.txt!");
}
else {
  Serial.println("Error, failed to open data file for writing!");
}

```

Again the **open** function is used but this time it's told to open the file for writing. In this mode the file will be opened for appending (i.e. data added to the end of it) if it exists, or it will be created if it doesn't exist. Once the file is open print functions like **print** and **println** can be used to write data to the file (just like writing to the serial monitor). Be sure to **close** the file when finished writing!

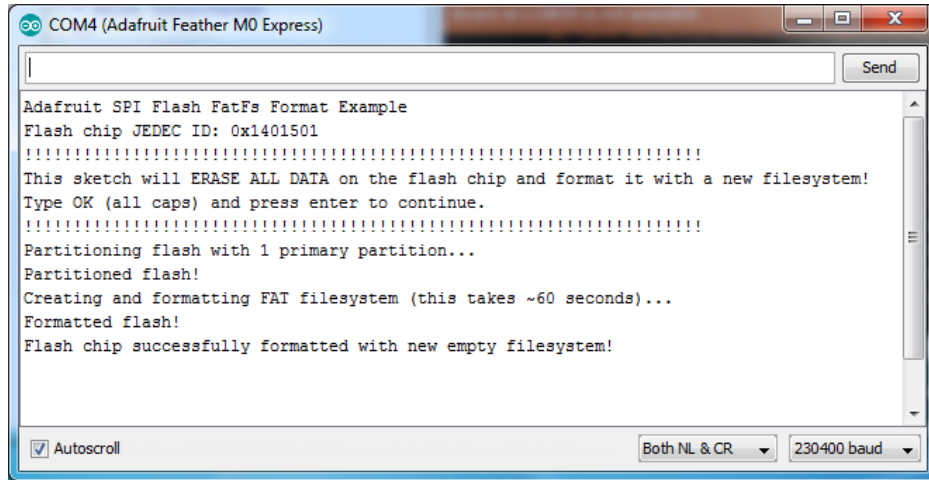
That's all there is to basic file reading and writing. Check out the **fatfs_full_usage** example for examples of even more functions like creating directories, deleting files & directories, checking the size of files, and more! Remember though to interact with CircuitPython files you need to use the **Adafruit_Feather_M0_CircuitPython** class as shown in the **fatfs_circuitpython** example above!

Format Flash Memory

The **fatfs_format** example will format the SPI flash with a new blank filesystem. **Be warned this sketch will delete all data on the flash memory, including any Python code or other data you might have stored!** The format sketch is useful if you'd like to wipe everything away and start fresh, or to help get back in a good state if the memory should get corrupted for some reason.

Be aware too the fatfs_format and examples below are not compatible with a CircuitPython-formatted flash chip! If you need to share data between Arduino & CircuitPython check out the **fatfs_circuitpython** example above.

To run the format sketch load it in the Arduino IDE and upload it to the M0 board. Then open the serial monitor at 115200 baud. You should see the serial monitor display a message asking you to confirm formatting the flash. If you don't see this message then close the serial monitor, press the board's reset button, and open the serial monitor again.



Type **OK** and press enter in the serial monitor input to confirm that you'd like to format the flash memory. **You need to enter OK in all capital letters!**

Once confirmed the sketch will format the flash memory. The format process takes about a minute so be patient as the data is erased and formatted. You should see a message printed once the format process is complete. At this point the flash chip will be ready to use with a brand new empty filesystem.

Datalogging Example

One handy use of the SPI flash is to store data, like datalogging sensor readings. The `fatfs_datalogging` example shows basic file writing/datalogging. Open the example in the Arduino IDE and upload it to your Feather M0 board.

Then open the serial monitor at 115200 baud. You should see a message printed every minute as the sketch writes a new line of data to a file on the flash filesystem.

To understand how to write to a file look in the loop function of the sketch:

```
// Open the datalogging file for writing. The FILE_WRITE mode will open
// the file for appending, i.e. it will add new data to the end of the file.
File dataFile = fatfs.open(FILE_NAME, FILE_WRITE);
// Check that the file opened successfully and write a line to it.
if (dataFile) {
  // Take a new data reading from a sensor, etc. For this example just
  // make up a random number.
  int reading = random(0,100);
  // Write a line to the file. You can use all the same print functions
  // as if you're writing to the serial monitor. For example to write
  // two CSV (commas separated) values:
  dataFile.print("Sensor #1");
  dataFile.print(",");
  dataFile.print(reading, DEC);
  dataFile.println();
  // Finally close the file when done writing. This is smart to do to make
  // sure all the data is written to the file.
  dataFile.close();
  Serial.println("Wrote new measurement to data file!");
}
```

Just like using the Arduino SD card library you create a **File** object by calling an **open** function and pointing it at the name of the file and how you'd like to open it (**FILE_WRITE** mode, i.e. writing new data to the end of the file). Notice

however instead of calling open on a global SD card object you're calling it on a **fatfs** object created earlier in the sketch (look at the top after the `#define` configuration values).

Once the file is opened it's simply a matter of calling **print** and **println** functions on the file object to write data inside of it. This is just like writing data to the serial monitor and you can print out text, numeric, and other types of data. Be sure to close the file when you're done writing to ensure the data is stored correctly!

Reading and Printing Files

The **fatfs_print_file** example will open a file (by default the `data.csv` file created by running the **fatfs_datalogging** example above) and print all of its contents to the serial monitor. Open the **fatfs_print_file** example and load it on your Feather M0 board, then open the serial monitor at 115200 baud. You should see the sketch print out the contents of `data.csv` (if you don't have a file called `data.csv` on the flash look at running the `datalogging` example above first).

To understand how to read data from a file look in the **setup** function of the sketch:

```
// Open the file for reading and check that it was successfully opened.
// The FILE_READ mode will open the file for reading.
File dataFile = fatfs.open(FILE_NAME, FILE_READ);
if (dataFile) {
  // File was opened, now print out data character by character until at the
  // end of the file.
  Serial.println("Opened file, printing contents below:");
  while (dataFile.available()) {
    // Use the read function to read the next character.
    // You can alternatively use other functions like readUntil, readString, etc.
    // See the fatfs_full_usage example for more details.
    char c = dataFile.read();
    Serial.print(c);
  }
}
```

Just like when writing data with the `datalogging` example you create a **File** object by calling the **open** function on a **fatfs** object. This time however you pass a file mode of **FILE_READ** which tells the filesystem you want to read data.

After you open a file for reading you can easily check if data is available by calling the **available** function on the file, and then read a single character with the **read** function. This makes it easy to loop through all of the data in a file by checking if it's available and reading a character at a time. However there are more advanced read functions you can use too--see the **fatfs_full_usage** example or even the [Arduino SD library documentation \(https://adafruit.it/ucu\)](https://adafruit.it/ucu) (the SPI flash library implements the same functions).

Full Usage Example

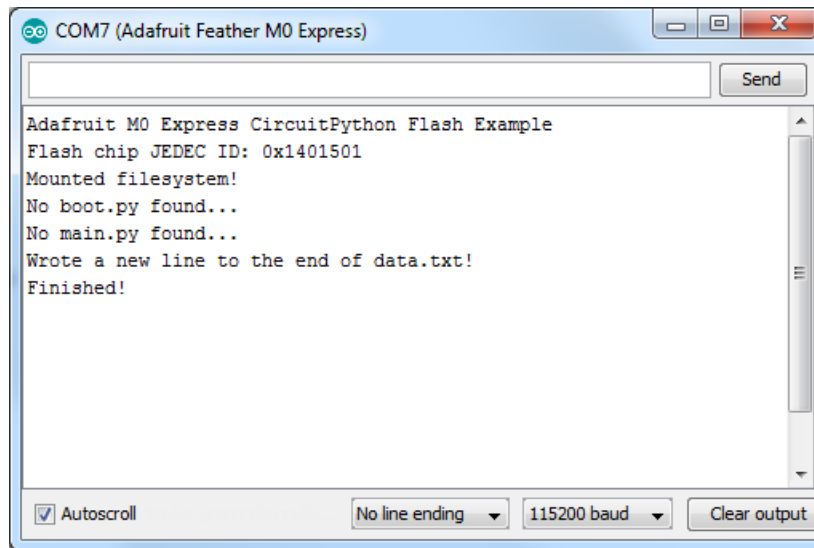
For a more complete demonstration of reading and writing files look at the **fatfs_full_usage** example. This examples uses every function in the library and demonstrates things like checking for the existence of a file, creating directories, deleting files, deleting directories, and more.

Remember the SPI flash library is built to have the same functions and interface as the [Arduino SD library \(https://adafruit.it/ucu\)](https://adafruit.it/ucu) so if you have code or examples that store data on a SD card they should be easy to adapt to use the SPI flash library, just create a **fatfs** object like in the examples above and use its **open** function instead of the global SD object's **open** function. Once you have a reference to a file all of the functions and usage should be the same between the SPI flash and SD libraries!

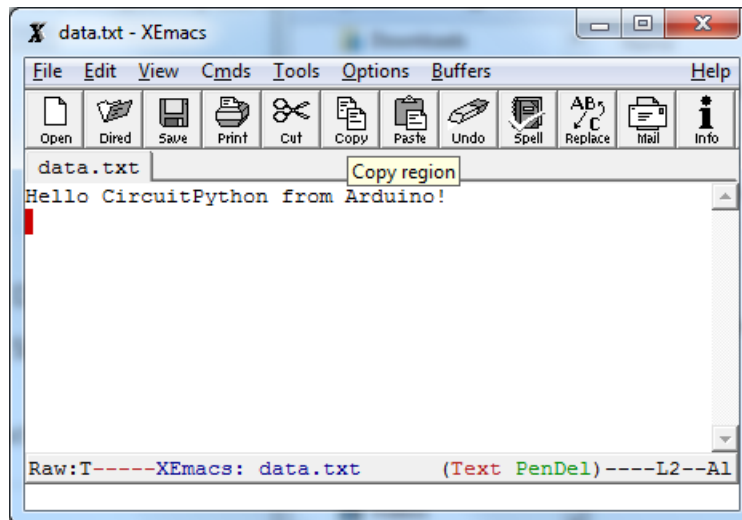
Accessing SPI Flash

Arduino doesn't have the ability to show up as a 'mass storage' disk drive. So instead we must use CircuitPython to do that part for us. Here's the full technique:

- Start the bootloader on the Express board. Drag over the latest **circuitpython** uf2 file
- After a moment, you should see a **CIRCUITPY** drive appear on your hard drive with **boot_out.txt** on it
- Now go to Arduino and upload the **fatfs_circuitpython** example sketch from the Adafruit SPI library. Open the serial console. It will successfully mount the filesystem and write a new line to **data.txt**



- Back on your computer, re-start the Express board bootloader, and re-drag **circuitpython.uf2** onto the **BOOT** drive to reinstall circuitpython
- Check the **CIRCUITPY** drive, you should now see **data.txt** which you can open to read!



Once you have your Arduino sketch working well, for datalogging, you can simplify this procedure by dragging **CURRENT.UF2** off of the **BOOT** drive to make a backup of the current program before loading circuitpython on. Then once you've accessed the file you want, re-drag **CURRENT.UF2** back onto the **BOOT** drive to re-install the Arduino sketch!

Feather HELP!

Even though this FAQ is labeled for Feather, the questions apply to ItsyBitsy's as well!

My ItsyBitsy/Feather stopped working when I unplugged the USB!

A lot of our example sketches have a

```
while (!Serial);
```

line in setup(), to keep the board waiting until the USB is opened. This makes it a lot easier to debug a program because you get to see all the USB data output. If you want to run your Feather without USB connectivity, delete or comment out that line

□ My Feather never shows up as a COM or Serial port in the Arduino IDE

A vast number of Itsy/Feather 'failures' are due to charge-only USB cables

We get upwards of 5 complaints a day that turn out to be due to charge-only cables!

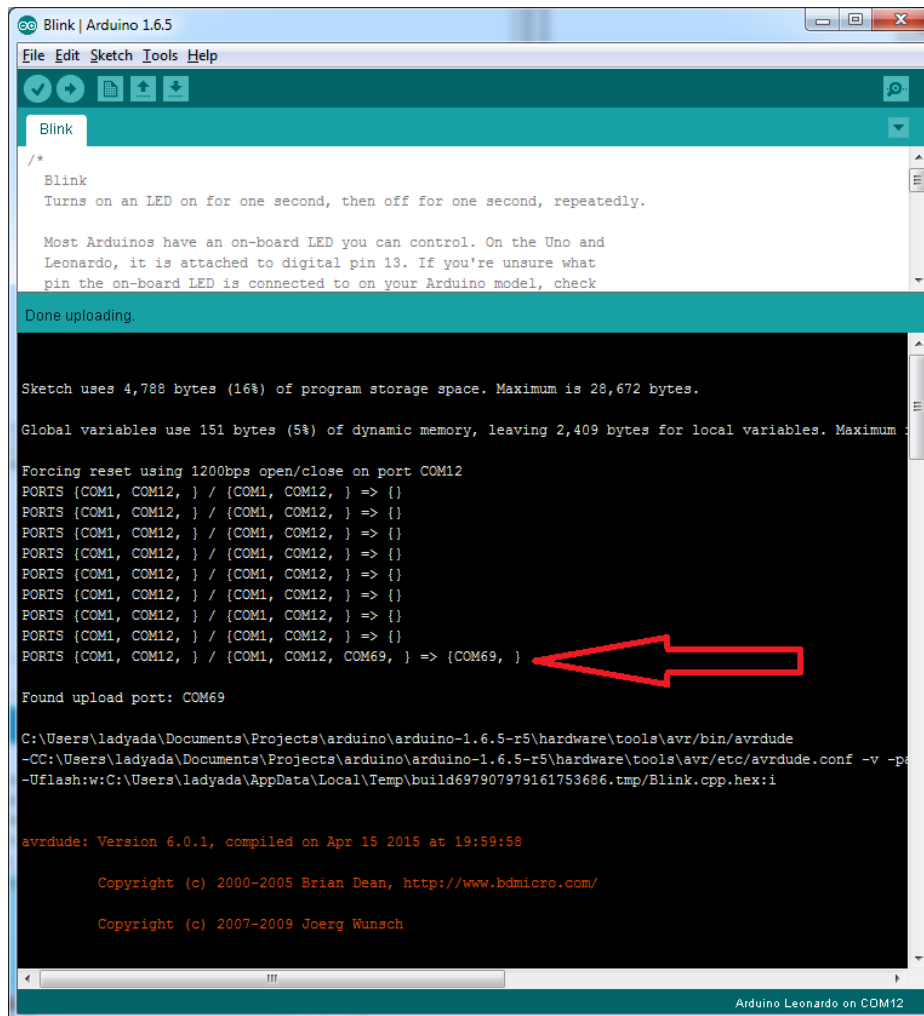
Use only a cable that you **know** is for data syncing

If you have any charge-only cables, cut them in half throw them out. We are serious! They tend to be low quality in general, and will only confuse you and others later, just get a good data+charge USB cable

□ Ack! I "did something" and now when I plug in the Itsy/Feather, it doesn't show up as a device anymore so I cant upload to it or fix it...

No problem! You can 'repair' a bad code upload easily. Note that this can happen if you set a watchdog timer or sleep mode that stops USB, or any sketch that 'crashes' your board

1. Turn on **verbose upload** in the Arduino IDE preferences
2. Plug in Itsy or Feather 32u4/M0, it won't show up as a COM/serial port that's ok
3. Open up the Blink example (Examples->Basics->Blink)
4. Select the correct board in the Tools menu, e.g. Feather 32u4, Feather M0, Itsy 32u4 or M0 (*physically check your board to make sure you have the right one selected!*)
5. Compile it (make sure that works)
6. Click Upload to attempt to upload the code
7. The IDE will print out a bunch of COM Ports as it tries to upload. **During this time, double-click the reset button, you'll see the red pulsing LED that tells you its now in bootloading mode**
8. The board will show up as the Bootloader COM/Serial port
9. The IDE should see the bootloader COM/Serial port and upload properly



❏ I can't get the Itsy/Feather USB device to show up - I get "USB Device Malfunctioning" errors!

This seems to happen when people select the wrong board from the Arduino Boards menu.

If you have a Feather 32u4 (look on the board to read what it is you have) Make sure you select **Feather 32u4** for ATMega32u4 based boards! Do not use anything else, do not use the 32u4 breakout board line.

If you have a Feather M0 (look on the board to read what it is you have) Make sure you select **Feather M0** - do not

use 32u4 or Arduino Zero

If you have a ItsyBitsy M0 (look on the board to read what it is you have) Make sure you select **ItsyBitsy M0** - do not use 32u4 or Arduino Zero

📌 I'm having problems with COM ports and my Itsy/Feather 32u4/M0

Theres **two** COM ports you can have with the 32u4/M0, one is the **user port** and one is the **bootloader port**. They are not the same COM port number!

When you upload a new user program it will come up with a user com port, particularly if you use Serial in your user

program.

If you crash your user program, or have a program that halts or otherwise fails, the user COM port can disappear.

When the user COM port disappears, Arduino will not be able to automatically start the bootloader and upload new software.

So you will need to help it by performing the click-during upload procedure to re-start the bootloader, and upload something that is known working like "Blink"

□ I don't understand why the COM port disappears, this does not happen on my Arduino UNO!

UNO-type Arduinos have a *seperate* serial port chip (aka "FTDI chip" or "Prolific PL2303" etc etc) which handles all serial port capability seperately than the main chip. This way if the main chip fails, you can always use the COM port.

M0 and 32u4-based Arduinos do not have a separate chip, instead the main processor performs this task for you. It allows for a lower cost, higher power setup...but requires a little more effort since you will need to 'kick' into the bootloader manually once in a while

📄 I'm trying to upload to my 32u4, getting "avrdude: butterfly_recv(): programmer is not responding" errors

This is likely because the bootloader is not kicking in and you are accidentally **trying to upload to the wrong COM port**

The best solution is what is detailed above: manually upload Blink or a similar working sketch by hand by manually launching the bootloader

I'm trying to upload to my Feather M0, and I get this error "Connecting to programmer: .avrdude: butterfly_recv(): programmer is not responding"

You probably don't have Feather M0 selected in the boards drop-down. Make sure you selected Feather M0.

□ I'm trying to upload to my Feather and i get this error "avrdude: ser_recv(): programmer is not responding"

You probably don't have Feather M0 / Feather 32u4 selected in the boards drop-down. Make sure you selected Feather M0 (or Feather 32u4).

□ I attached some wings to my Feather and now I can't read the battery voltage!

Make sure your Wing doesn't use pin #9 which is the analog sense for the lipo battery!

□ The yellow LED is flickering on my Feather, but no battery is plugged in, why is that?

The charge LED is automatically driven by the Lipoly charger circuit. It will try to detect a battery and is expecting one to be attached. If there isn't one it may flicker once in a while when you use power because it's trying to charge a (non-existent) battery.

It's not harmful, and it's totally normal!

What is CircuitPython?

CircuitPython is a programming language designed to simplify experimenting and learning to program on low-cost microcontroller boards. It makes getting started easier than ever with no upfront desktop downloads needed. Once you get your board set up, open any text editor, and get started editing code. It's that simple.



CircuitPython is based on Python

Python is the fastest growing programming language. It's taught in schools and universities. It's a high-level programming language which means it's designed to be easier to read, write and maintain. It supports modules and packages which means it's easy to reuse your code for other projects. It has a built in interpreter which means there are no extra steps, like *compiling*, to get your code to work. And of course, Python is Open Source Software which means it's free for anyone to use, modify or improve upon.

CircuitPython adds hardware support to all of these amazing features. If you already have Python knowledge, you can easily apply that to using CircuitPython. If you have no previous experience, it's really simple to get started!



Why would I use CircuitPython?

CircuitPython is designed to run on microcontroller boards. A microcontroller board is a board with a microcontroller chip that's essentially an itty-bitty all-in-one computer. The board you're holding is a microcontroller board! CircuitPython is easy to use because all you need is that little board, a USB cable, and a computer with a USB connection. But that's only the beginning.

Other reasons to use CircuitPython include:

- **You want to get up and running quickly.** Create a file, edit your code, save the file, and it runs immediately. There is no compiling, no downloading and no uploading needed.

- **You're new to programming.** CircuitPython is designed with education in mind. It's easy to start learning how to program and you get immediate feedback from the board.
- **Easily update your code.** Since your code lives on the disk drive, you can edit it whenever you like, you can also keep multiple files around for easy experimentation.
- **The serial console and REPL.** These allow for live feedback from your code and interactive programming.
- **File storage.** The internal storage for CircuitPython makes it great for data-logging, playing audio clips, and otherwise interacting with files.
- **Strong hardware support.** There are many libraries and drivers for sensors, breakout boards and other external components.
- **It's Python!** Python is the fastest-growing programming language. It's taught in schools and universities. CircuitPython is almost-completely compatible with Python. It simply adds hardware support.

This is just the beginning. CircuitPython continues to evolve, and is constantly being updated. We welcome and encourage feedback from the community, and we incorporate this into how we are developing CircuitPython. That's the core of the open source concept. This makes CircuitPython better for you and everyone who uses it!

CircuitPython



As we continue to develop CircuitPython and create new releases, we will stop supporting older releases. If you are running CircuitPython 2.x, you need to update to 3.x: <https://learn.adafruit.com/welcome-to-circuitpython/installing-circuitpython>

CircuitPython (<https://adafru.it/tB7>) is a derivative of MicroPython (<https://adafru.it/BeZ>) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

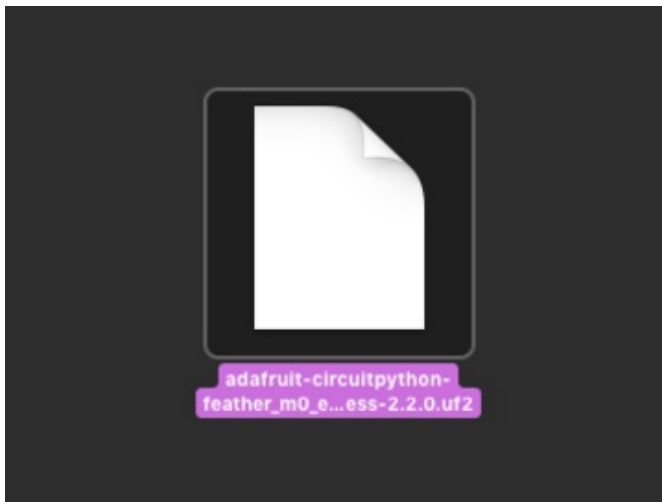
The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

Set up CircuitPython Quick Start!

Follow this quick step-by-step for super-fast Python power :)

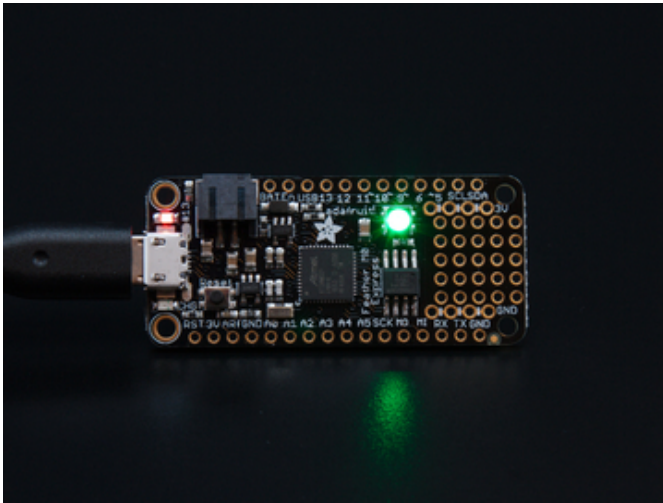
<https://adafru.it/Em9>

<https://adafru.it/Em9>



Click the link above and download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).

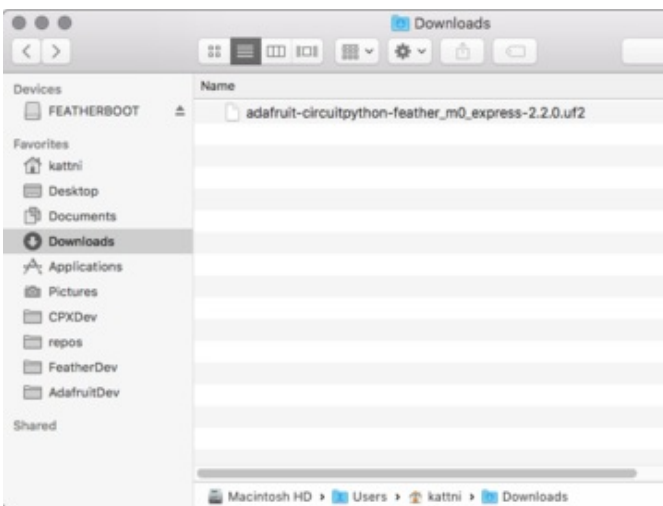


Plug your Feather M0 into your computer using a known-good USB cable.

A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

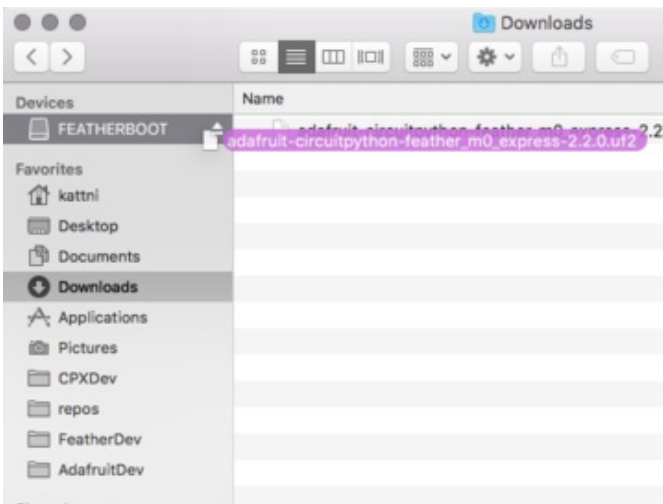
Double-click the **Reset** button next to the USB connector on your board, and you will see the NeoPixel RGB LED turn green. If it turns red, check the USB cable, try another USB port, etc. **Note:** The little red LED next to the USB connector will pulse red. That's ok!

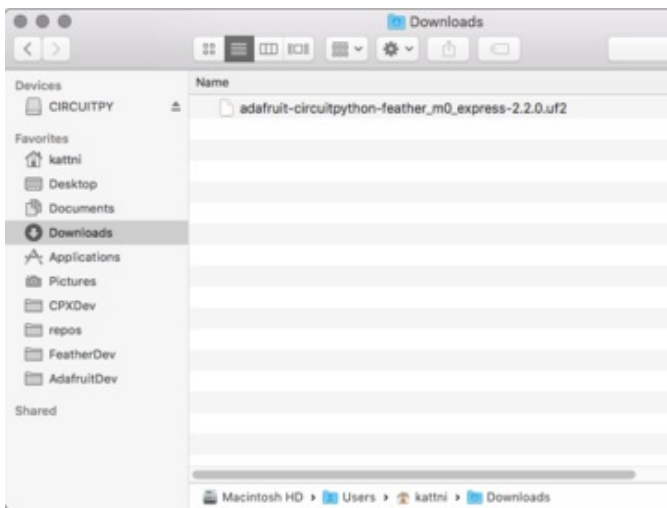
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!



You will see a new disk drive appear called **FEATHERBOOT**.

Drag the `adafruit_circuitpython_etc.uf2` file to **FEATHERBOOT**.





The LED will flash. Then, the **FEATHERBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it, you're done! :)

Further Information

For more detailed info on installing CircuitPython, check out [Installing CircuitPython \(https://adafru.it/Amd\)](https://adafru.it/Amd).

Installing Mu Editor

Mu is a simple code editor that works with the Adafruit CircuitPython boards. It's written in Python and works on Windows, MacOS, Linux and Raspberry Pi. The serial console is built right in so you get immediate feedback from your board's serial output!



Mu is our recommended editor - please use it (unless you are an experienced coder with a favorite editor already!)

Download and Install Mu



Download Mu

from <https://codewith.mu> (<https://adafru.it/Be6>). Click the **Download** or **Start Here** links there for downloads and installation instructions. The website has a wealth of other information, including extensive tutorials and and how-to's.

Using Mu



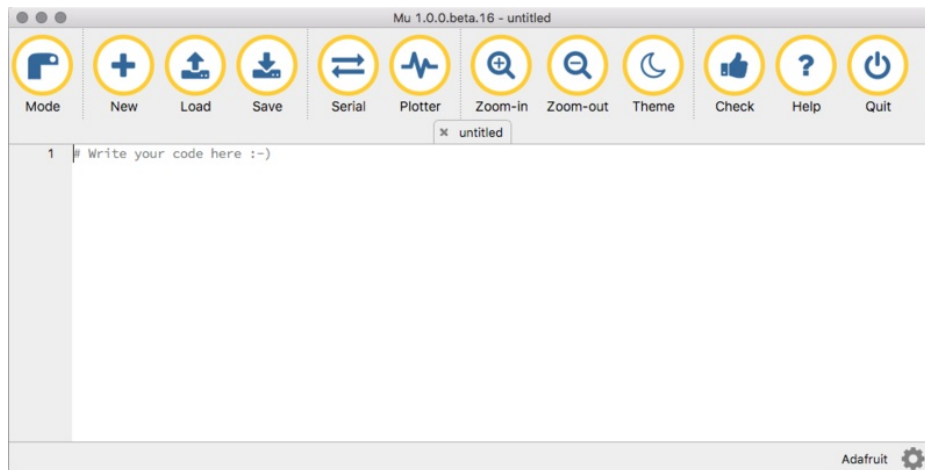
The first time you start Mu, you will be prompted to select your 'mode' - you can always change your mind later. For now please select **Adafruit!**

The current mode is displayed in the lower right corner of the window, next to the "gear" icon. If the mode says "Microbit" or something else, click on that and then choose "Adafruit" in the dialog box that appears.



Mu attempts to auto-detect your board, so please plug in your CircuitPython device and make sure it shows up as a **CIRCUITPY** drive before starting Mu

Now you're ready to code! Lets keep going....



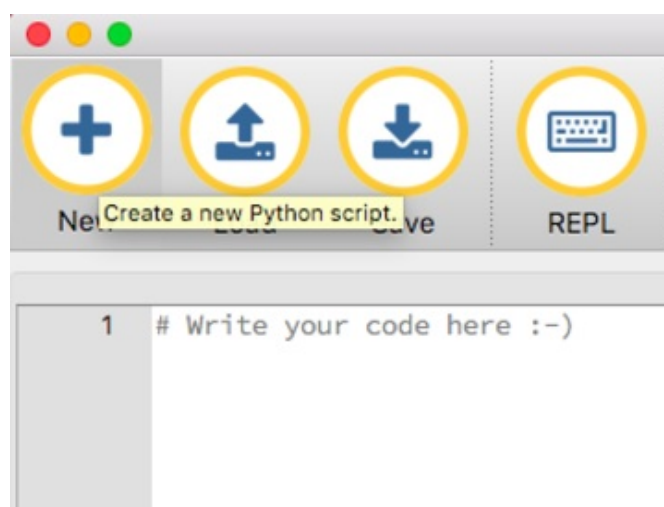
Creating and Editing Code

One of the best things about CircuitPython is how simple it is to get code up and running. In this section, we're going to cover how to create and edit your first CircuitPython program.

To create and edit code, all you'll need is an editor. There are many options. **We strongly recommend using Mu! It's designed for CircuitPython, and it's really simple and easy to use, with a built in serial console!**

If you don't or can't use Mu, there are basic text editors built into every operating system such as Notepad on Windows, TextEdit on Mac, and gedit on Linux. However, many of these editors don't write back changes immediately to files that you edit. That can cause problems when using CircuitPython. See the [Editing Code \(https://adafru.it/id3\)](https://adafru.it/id3) section below. If you want to skip that section for now, make sure you do "Eject" or "Safe Remove" on Windows or "sync" on Linux after writing a file if you aren't using Mu. (This is not a problem on MacOS.)

Creating Code



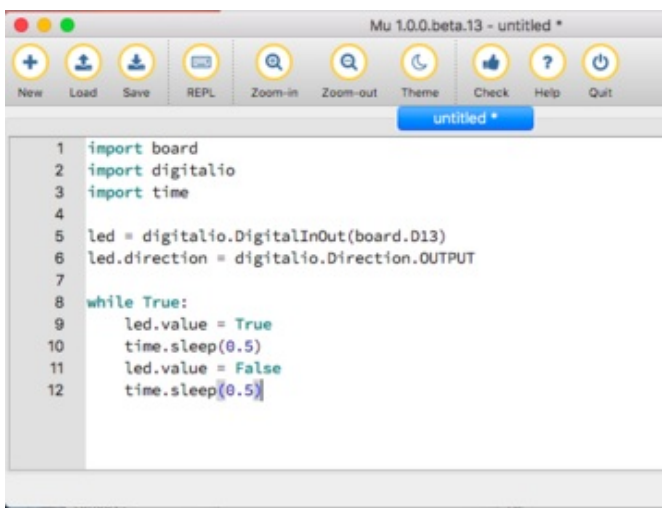
Open your editor, and create a new file. If you are using Mu, click the **New** button in the top left

Copy and paste the following code into your editor:

```
import board
import digitalio
import time

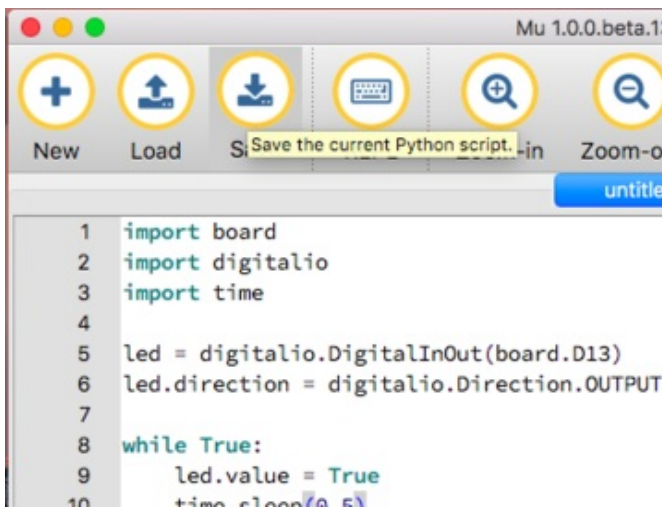
led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```



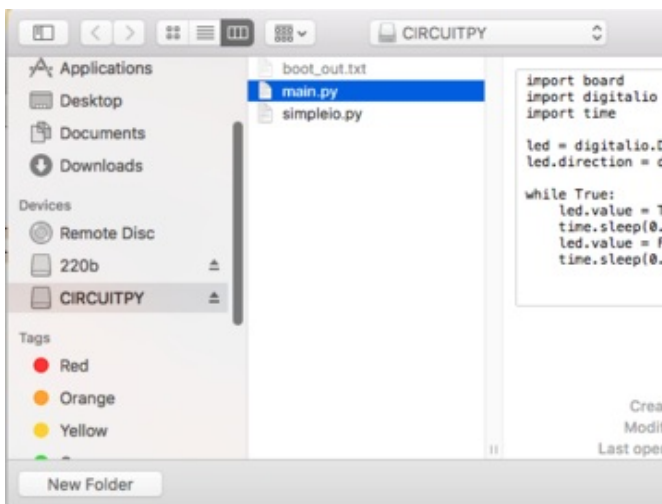
```
1 import board
2 import digitalio
3 import time
4
5 led = digitalio.DigitalInOut(board.D13)
6 led.direction = digitalio.Direction.OUTPUT
7
8 while True:
9     led.value = True
10    time.sleep(0.5)
11    led.value = False
12    time.sleep(0.5)
```

It will look like this - note that under the `while True:` line, the next four lines have spaces to indent them, but they're indented exactly the same amount. All other lines have no spaces before the text.



```
1 import board
2 import digitalio
3 import time
4
5 led = digitalio.DigitalInOut(board.D13)
6 led.direction = digitalio.Direction.OUTPUT
7
8 while True:
9     led.value = True
10    time.sleep(0.5)
```

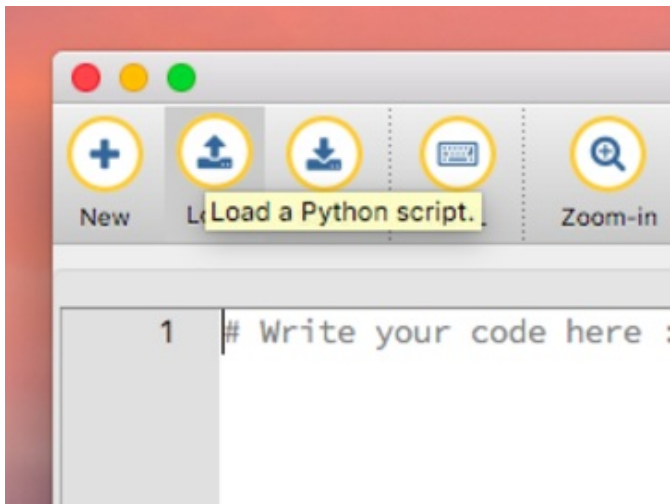
Save this file as `code.py` on your CIRCUITPY drive.



On each board you'll find a tiny red LED. It should now be blinking. Once per second

Congratulations, you've just run your first CircuitPython program!

Editing Code



To edit code, open the **code.py** file on your CIRCUITPY drive into your editor.

Make the desired changes to your code. Save the file. That's it!

Your code changes are run as soon as the file is done saving.

There's just one warning we have to give you before we continue...

 Don't Click Reset or Unplug!

The CircuitPython code on your board detects when the files are changed or written and will automatically re-start your code. This makes coding very fast because you save, and it re-runs.

However, you must wait until the file is done being saved before unplugging or resetting your board! On Windows using some editors this can sometimes take up to 90 seconds, on Linux it can take 30 seconds to complete because the text editor does not save the file completely. Mac OS does not seem to have this delay, which is nice!

This is really important to be aware of. If you unplug or reset the board before your computer finishes writing the file to your board, you can corrupt the drive. If this happens, you may lose the code you've written, so it's important to backup your code to your computer regularly.

There are a few ways to avoid this:

1. Use an editor that writes out the file completely when you save it.

Recommended editors:

- **mu** (<https://adafru.it/Be6>) is an editor that safely writes all changes (it's also our recommended editor!)
- **emacs** (<https://adafru.it/xNA>) is also an editor that will **fully write files on save** (<https://adafru.it/Be7>)
- **Sublime Text** (<https://adafru.it/xNB>) safely writes all changes
- **Visual Studio Code** (<https://adafru.it/Be9>) appears to safely write all changes
- **gedit** on Linux appears to safely write all changes

Recommended *only* with particular settings or with add-ons:

- **vim** (<https://adafru.it/ek9>) / **vi** safely writes all changes. But set up **vim** to not write **swapfiles** (<https://adafru.it/ELO>) (.swp files: temporary records of your edits) to CIRCUITPY. Run vim with `vim -n`, set the `no swapfile` option, or set the `directory` option to write swapfiles elsewhere. Otherwise the swapfile writes trigger restarts of your program.
- The **PyCharm IDE** (<https://adafru.it/xNC>) is safe if "Safe Write" is turned on in Settings->System Settings->Synchronization (true by default).
- If you are using **Atom** (<https://adafru.it/fMG>), install the **fsync-on-save package** (<https://adafru.it/E9m>) so that it will always write out all changes to files on **CIRCUITPY**.
- **SlickEdit** (<https://adafru.it/DdP>) works only if you **add a macro to flush the disk** (<https://adafru.it/ven>).

We *don't* recommend these editors:

- **notepad** (the default Windows editor) and **Notepad++** can be slow to write, so we recommend the editors above! If you are using notepad, be sure to eject the drive (see below)
- **IDLE** does not force out changes immediately
- **nano** (on Linux) does not force out changes
- **Anything else** - we haven't tested other editors so please use a recommended one!

2. Eject or Sync the Drive After Writing

If you are using one of our not-recommended-editors, not all is lost! You can still make it work.

On Windows, you can **Eject** or **Safe Remove** the CIRCUITPY drive. It won't actually eject, but it will force the operating system to save your file to disk. On Linux, use the `sync` command in a terminal to force the write to disk.

☐ Oh No I Did Something Wrong and Now The CIRCUITPY Drive Doesn't Show Up!!!

Don't worry! Corrupting the drive isn't the end of the world (or your board!). If this happens, follow the steps found on the [Troubleshooting](#) page of every board guide to get your board up and running again.

Back to Editing Code...

Now! Let's try editing the program you added to your board. Open your `code.py` file into your editor. We'll make a simple change. Change the first `0.5` to `0.1`. The code should look like this:

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.5)
```

Leave the rest of the code as-is. Save your file. See what happens to the LED on your board? Something changed! Do you know why? Let's find out!

Exploring Your First CircuitPython Program

First, we'll take a look at the code we're editing.

Here is the original code again:

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Imports & Libraries

Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. These files are called **libraries**. Some of them are built into CircuitPython. Others are stored on your CIRCUITPY drive in a folder called **lib**.

```
import board
import digitalio
import time
```

The **import** statements tells the board that you're going to use a particular library in your code. In this example, we imported three libraries: **board**, **digitalio**, and **time**. All three of these libraries are built into CircuitPython, so no separate files are needed. That's one of the things that makes this an excellent first example. You don't need anything extra to make it work! **board** gives you access to the *hardware on your board*, **digitalio** lets you *access that hardware as inputs/outputs* and **time** lets you pass time by 'sleeping'

Setting Up The LED

The next two lines setup the code to use the LED.

```
led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT
```

Your board knows the red LED as **D13**. So, we initialise that pin, and we set it to output. We set **led** to equal the rest of that information so we don't have to type it all out again later in our code.

Loop-de-loops

The third section starts with a **while** statement. **while True:** essentially means, "forever do the following:". **while True:** creates a loop. Code will loop "while" the condition is "true" (vs. false), and as **True** is never False, the code will loop forever. All code that is indented under **while True:** is "inside" the loop.

Inside our loop, we have four items:

```
while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

First, we have `led.value = True`. This line tells the LED to turn on. On the next line, we have `time.sleep(0.5)`. This line is telling CircuitPython to pause running code for 0.5 seconds. Since this is between turning the led on and off, the led will be on for 0.5 seconds.

The next two lines are similar. `led.value = False` tells the LED to turn off, and `time.sleep(0.5)` tells CircuitPython to pause for another 0.5 seconds. This occurs between turning the led off and back on so the LED will be off for 0.5 seconds too.

Then the loop will begin again, and continue to do so as long as the code is running!

So, when you changed the first `0.5` to `0.1`, you decreased the amount of time that the code leaves the LED on. So it blinks on really quickly before turning off!

Great job! You've edited code in a CircuitPython program!

□ What if I don't have the loop?

If you don't have the loop, the code will run to the end and exit. This can lead to some unexpected behavior in simple programs like this since the "exit" also resets the state of the hardware. This is a different behavior than running commands via REPL. So if you are writing a simple program that doesn't seem to work, you may need to add a loop to the end so the program doesn't exit.

The simplest loop would be:

```
while True:
    pass
```

And remember - you can press `<CTRL><C>` to exit the loop.

See also the [Behavior section in the docs](#).

More Changes

We don't have to stop there! Let's keep going. Change the second `0.5` to `0.1` so it looks like this:

```
while True:
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.1)
```

Now it blinks really fast! You decreased the both time that the code leaves the LED on and off!

Now try increasing both of the `0.1` to `1`. Your LED will blink much more slowly because you've increased the amount of time that the LED is turned on and off.

Well done! You're doing great! You're ready to start into new examples and edit them to see what happens! These were simple changes, but major changes are done using the same process. Make your desired change, save it, and get the results. That's really all there is to it!

Naming Your Program File

CircuitPython looks for a code file on the board to run. There are four options: `code.txt`, `code.py`, `main.txt` and `main.py`. CircuitPython looks for those files, in that order, and then runs the first one it finds. While we suggest using `code.py` as your code file, it is important to know that the other options exist. If your program doesn't seem to be updating as you work, make sure you haven't created another code file that's being read instead of the one you're working on.

Connecting to the Serial Console

One of the staples of CircuitPython (and programming in general!) is something called a "print statement". This is a line you include in your code that causes your code to output text. A print statement in CircuitPython looks like this:

```
print("Hello, world!")
```

This line would result in:

```
Hello, world!
```

However, these print statements need somewhere to display. That's where the serial console comes in!

The serial console receives output from your CircuitPython board sent over USB and displays it so you can see it. This is necessary when you've included a print statement in your code and you'd like to see what you printed. It is also helpful for troubleshooting errors, because your board will send errors and the serial console will print those too.

The serial console requires a terminal program. A terminal is a program that gives you a text-based interface to perform various tasks.



If you're on Linux, and are seeing multi-second delays connecting to the serial console, or are seeing "AT" and other gibberish when you connect, then the modemmanager service might be interfering. Just remove it; it doesn't have much use unless you're still using dial-up modems. To remove, type this command at a shell:

```
sudo apt purge modemmanager
```

Are you using Mu?

If so, good news! The serial console is **built into Mu** and will **autodetect your board** making using the REPL *really really easy*.

Please note that Mu does yet not work with nRF52 or ESP8266-based CircuitPython boards, skip down to the next section for details on using a terminal program.



First, make sure your CircuitPython board is plugged in. If you are using Windows 7, make sure you installed the drivers (<https://adafru.it/Amd>).

Once in Mu, look for the **Serial** button in the menu and click it.



Setting Permissions on Linux

On Linux, if you see an error box something like the one below when you press the **Serial** button, you need to add yourself to a user group to have permission to connect to the serial console.



On Ubuntu and Debian, add yourself to the **dialout** group by doing:

```
sudo adduser $USER dialout
```

After running the command above, reboot your machine to gain access to the group. On other Linux distributions, the group you need may be different. See [Advanced Serial Console on Mac and Linux \(https://adafru.it/AAI\)](https://adafru.it/AAI) for details on how to add yourself to the right group.

Using Something Else?

If you're not using Mu to edit, are using ESP8266 or nRF52 CircuitPython, or if for some reason you are not a fan of the built in serial console, you can run the serial console as a separate program.

[Windows requires you to download a terminal program, check out this page for more details \(https://adafru.it/AAH\)](https://adafru.it/AAH)

[Mac and Linux both have one built in, though other options are available for download, check this page for more details \(https://adafru.it/AAI\)](https://adafru.it/AAI)

Interacting with the Serial Console

Once you've successfully connected to the serial console, it's time to start using it.

The code you wrote earlier has no output to the serial console. So, we're going to edit it to create some output.

Open your code.py file into your editor, and include a `print` statement. You can print anything you like! Just include your phrase between the quotation marks inside the parentheses. For example:

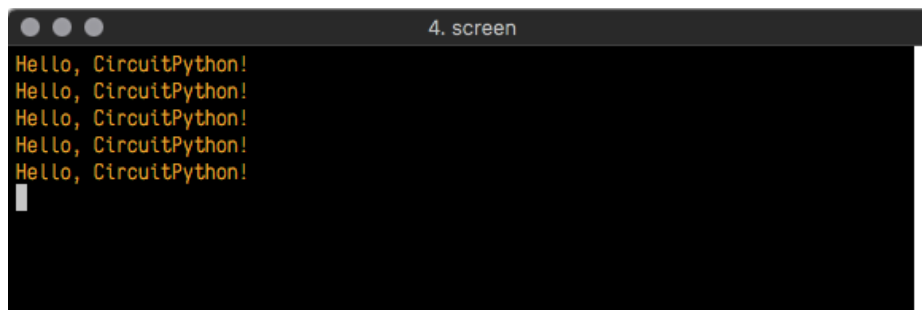
```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

while True:
    print("Hello, CircuitPython!")
    led.value = True
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

Save your file.

Now, let's go take a look at the window with our connection to the serial console.



```
4. screen
Hello, CircuitPython!
Hello, CircuitPython!
Hello, CircuitPython!
Hello, CircuitPython!
Hello, CircuitPython!
```

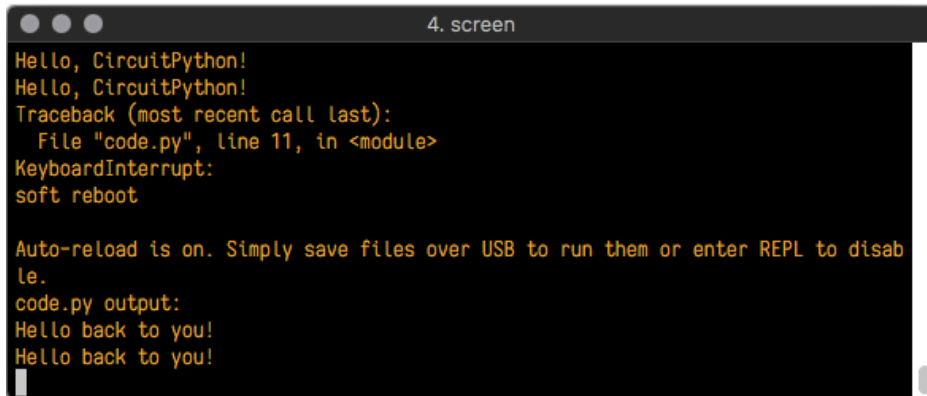
Excellent! Our print statement is showing up in our console! Try changing the printed text to something else.



```
code.py
1 import board
2 import digitalio
3 import time
4
5 led = digitalio.DigitalInOut(board.D13)
6 led.direction = digitalio.Direction.OUTPUT
7
8 while True:
9     print("Hello back to you!")
10    led.value = True
11    time.sleep(1)
12    led.value = False
13    time.sleep(1)
14
```

Keep your serial console window where you can see it. Save your file. You'll see what the serial console displays when

the board reboots. Then you'll see your new change!



```
4. screen
Hello, CircuitPython!
Hello, CircuitPython!
Traceback (most recent call last):
  File "code.py", line 11, in <module>
KeyboardInterrupt:
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Hello back to you!
Hello back to you!
```

The **Traceback (most recent call last):** is telling you the last thing your board was doing before you saved your file. This is normal behavior and will happen every time the board resets. This is really handy for troubleshooting. Let's introduce an error so we can see how it is used.

Delete the **e** at the end of **True** from the line **led.value = True** so that it says **led.value = Tru**



```
code.py
1 import board
2 import digitalio
3 import time
4
5 led = digitalio.DigitalInOut(board.D13)
6 led.direction = digitalio.Direction.OUTPUT
7
8 while True:
9     print("Hello back to you!")
10    led.value = Tru
11    time.sleep(1)
12    led.value = False
13    time.sleep(1)
14
```

Save your file. You will notice that your red LED will stop blinking, and you may have a colored status LED blinking at you. This is because the code is no longer correct and can no longer run properly. We need to fix it!

Usually when you run into errors, it's not because you introduced them on purpose. You may have 200 lines of code, and have no idea where your error could be hiding. This is where the serial console can help. Let's take a look!

```
5. screen
Hello back to you!
Traceback (most recent call last):
  File "code.py", line 13, in <module>
KeyboardInterrupt:
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Hello back to you!
Traceback (most recent call last):
  File "code.py", line 10, in <module>
NameError: name 'Tru' is not defined

Press any key to enter the REPL. Use CTRL-D to reload.
```

The **Traceback (most recent call last):** is telling you that the last thing it was able to run was line 10 in your code. The next line is your error: **NameError: name 'Tru' is not defined**. This error might not mean a lot to you, but combined with knowing the issue is on line 10, it gives you a great place to start!

Go back to your code, and take a look at line 10. Obviously, you know what the problem is already. But if you didn't, you'd want to look at line 10 and see if you could figure it out. If you're still unsure, try googling the error to get some help. In this case, you know what to look for. You spelled True wrong. Fix the typo and save your file.

```
5. screen
le.
code.py output:
Hello back to you!
Traceback (most recent call last):
  File "code.py", line 10, in <module>
NameError: name 'Tru' is not defined

Press any key to enter the REPL. Use CTRL-D to reload.
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Hello back to you!
Hello back to you!
```

Nice job fixing the error! Your serial console is streaming and your red LED is blinking again.

The serial console will display any output generated by your code. Some sensors, such as a humidity sensor or a thermistor, receive data and you can use print statements to display that information. You can also use print statements for troubleshooting. If your code isn't working, and you want to know where it's failing, you can put print statements in various places to see where it stops printing.

The serial console has many uses, and is an amazing tool overall for learning and programming!

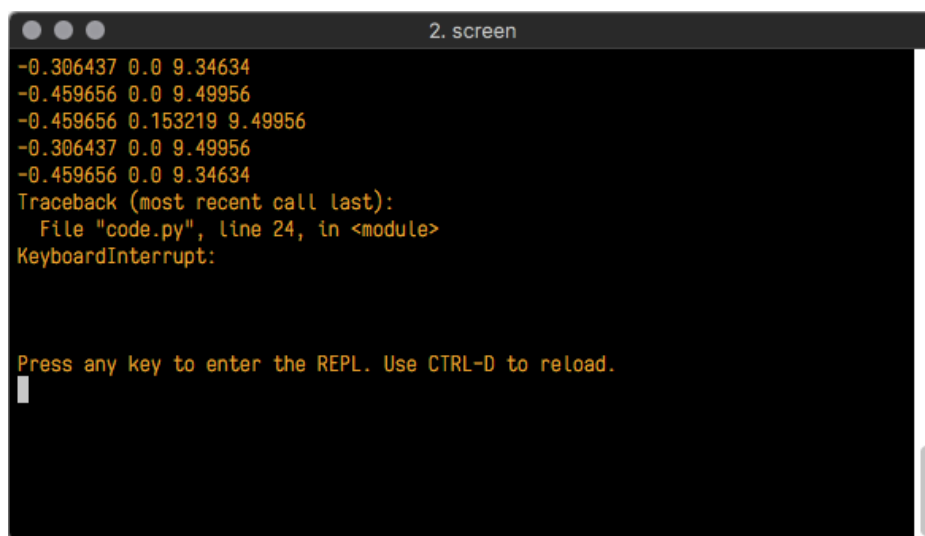
The REPL

The other feature of the serial connection is the **Read-Evaluate-Print-Loop**, or REPL. The REPL allows you to enter individual lines of code and have them run immediately. It's really handy if you're running into trouble with a particular program and can't figure out why. It's interactive so it's great for testing new ideas.

To use the REPL, you first need to be connected to the serial console. Once that connection has been established, you'll want to press **Ctrl + C**.

If there is code running, it will stop and you'll see **Press any key to enter the REPL. Use CTRL-D to reload**. Follow those instructions, and press any key on your keyboard.

The **Traceback (most recent call last)**: is telling you the last thing your board was doing before you pressed Ctrl + C and interrupted it. The **KeyboardInterrupt** is you pressing Ctrl + C. This information can be handy when troubleshooting, but for now, don't worry about it. Just note that it is expected behavior.



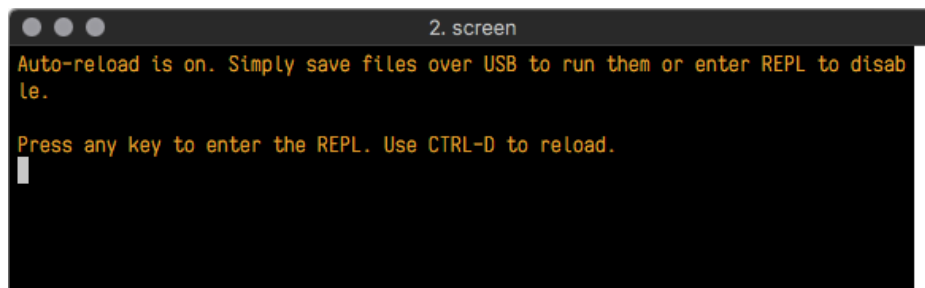
```

2. screen
-0.306437 0.0 9.34634
-0.459656 0.0 9.49956
-0.459656 0.153219 9.49956
-0.306437 0.0 9.49956
-0.459656 0.0 9.34634
Traceback (most recent call last):
  File "code.py", line 24, in <module>
KeyboardInterrupt:

Press any key to enter the REPL. Use CTRL-D to reload.

```

If there is no code running, you will enter the REPL immediately after pressing Ctrl + C. There is no information about what your board was doing before you interrupted it because there is no code running.



```

2. screen
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

Press any key to enter the REPL. Use CTRL-D to reload.

```

Either way, once you press a key you'll see a **>>>** prompt welcoming you to the REPL!

```
2. screen
Adafruit CircuitPython 2.1.0 on 2017-10-17; Adafruit CircuitPlayground Express w
ith samd21g18
>>> |
```

If you have trouble getting to the `>>>` prompt, try pressing `Ctrl + C` a few more times.

The first thing you get from the REPL is information about your board.

```
Adafruit CircuitPython 2.1.0 on 2017-10-17; Adafruit CircuitPlayground Express with samd21g18
```

This line tells you the version of CircuitPython you're using and when it was released. Next, it gives you the type of board you're using and the type of microcontroller the board uses. Each part of this may be different for your board depending on the versions you're working with.

This is followed by the CircuitPython prompt.

```
>>> |
```

From this prompt you can run all sorts of commands and code. The first thing we'll do is run `help()`. This will tell us where to start exploring the REPL. To run code in the REPL, type it in next to the REPL prompt.

Type `help()` next to the prompt in the REPL.

```
Adafruit CircuitPython 2.1.0 on 2017-10-17; Adafruit Feather M0 Express with samd21
g18
>>> help() |
```

Then press enter. You should then see a message.

```
2. screen
Auto-reload is on. Simply save files over USB to run them or enter REPL to disab
le.

Press any key to enter the REPL. Use CTRL-D to reload.

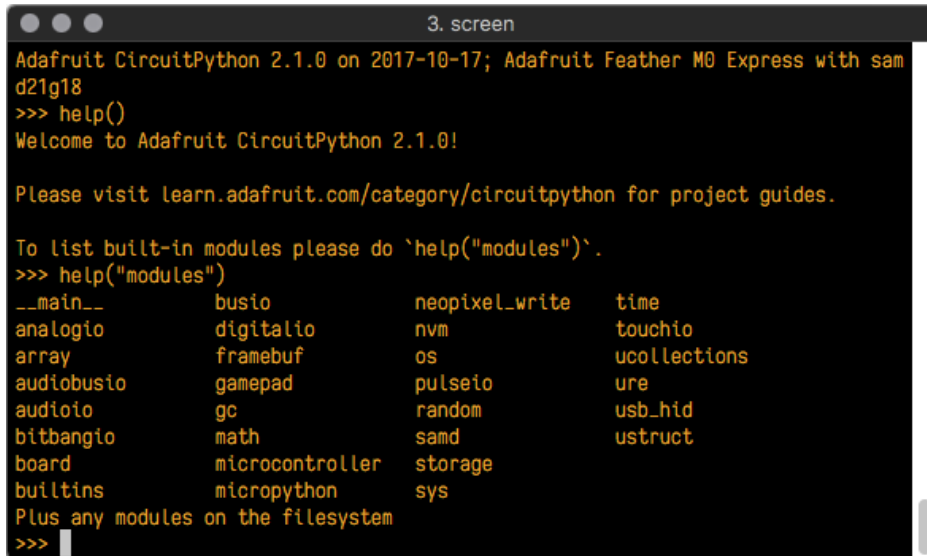
Adafruit CircuitPython 2.1.0 on 2017-10-17; Adafruit CircuitPlayground Express w
ith samd21g18
>>> help()
Welcome to Adafruit CircuitPython 2.1.0!

Please visit learn.adafruit.com/category/circuitpython for project guides.

To list built-in modules please do `help("modules")`.
>>> |
```

First part of the message is another reference to the version of CircuitPython you're using. Second, a URL for the CircuitPython related project guides. Then... wait. What's this? `To list built-in modules, please do `help("modules")`.` Remember the libraries you learned about while going through creating code? That's exactly what this is talking about! This is a perfect place to start. Let's take a look!

Type `help("modules")` into the REPL next to the prompt, and press enter.




```
3. screen
Adafruit CircuitPython 2.1.0 on 2017-10-17; Adafruit Feather M0 Express with sam
d21g18
>>> help()
Welcome to Adafruit CircuitPython 2.1.0!

Please visit learn.adafruit.com/category/circuitpython for project guides.

To list built-in modules please do `help("modules")`.
>>> help("modules")
__main__      busio          neopixel_write  time
analogio      digitalio     nvm              touchio
array         framebuffer   os               ucollections
audiobusio    gamepad       pulseio          ure
audioio       gc            random           usb_hid
bitbangio     math          samd             ustruct
board         microcontroller storage
builtins      micropython   sys
Plus any modules on the filesystem
>>>
```

This is a list of all the core libraries built into CircuitPython. We discussed how `board` contains all of the pins on the board that you can use in your code. From the REPL, you are able to see that list!

Type `import board` into the REPL and press enter. It'll go to a new prompt. It might look like nothing happened, but that's not the case! If you recall, the `import` statement simply tells the code to expect to do something with that module. In this case, it's telling the REPL that you plan to do something with that module.



```
3. screen
d21g18
>>> help()
Welcome to Adafruit CircuitPython 2.1.0!

Please visit learn.adafruit.com/category/circuitpython for project guides.

To list built-in modules please do `help("modules")`.
>>> help("modules")
__main__      busio          neopixel_write  time
analogio      digitalio     nvm              touchio
array         framebuffer   os               ucollections
audiobusio    gamepad       pulseio          ure
audioio       gc            random           usb_hid
bitbangio     math          samd             ustruct
board         microcontroller storage
builtins      micropython   sys
Plus any modules on the filesystem
>>> import board
>>>
```

Next, type `dir(board)` into the REPL and press enter.

```
3. screen

Please visit learn.adafruit.com/category/circuitpython for project guides.

To list built-in modules please do `help("modules")`.
>>> help("modules")
__main__      busio          neopixel_write  time
analogio      digitalio     nvm              touchio
array         framebuffer   os               ucollections
audiobusio    gamepad       pulseio          ure
audioio       gc            random           usb_hid
bitbangio     math          samd             ustruct
board         microcontroller  storage
builtins      micropython   sys
Plus any modules on the filesystem
>>> import board
>>> dir(board)
['A0', 'A1', 'A2', 'A3', 'A4', 'A5', 'SCK', 'MOSI', 'MISO', 'D0', 'RX', 'D1', 'TX',
 'SDA', 'SCL', 'D5', 'D6', 'D9', 'D10', 'D11', 'D12', 'D13', 'NEOPIXEL']
>>>
```

This is a list of all of the pins on your board that are available for you to use in your code. Each board's list will differ slightly depending on the number of pins available. Do you see [D13](#) ? That's the pin you used to blink the red LED!

The REPL can also be used to run code. Be aware that **any code you enter into the REPL isn't saved** anywhere. If you're testing something new that you'd like to keep, make sure you have it saved somewhere on your computer as well!

Every programmer in every programming language starts with a piece of code that says, "Hello, World." We're going to say hello to something else. Type into the REPL:

```
print("Hello, CircuitPython!")
```

Then press enter.

```
>>> print("Hello, CircuitPython!")
Hello, CircuitPython!
>>>
```

That's all there is to running code in the REPL! Nice job!

You can write single lines of code that run stand-alone. You can also write entire programs into the REPL to test them. As we said though, remember that nothing typed into the REPL is saved.

There's a lot the REPL can do for you. It's great for testing new ideas if you want to see if a few new lines of code will work. It's fantastic for troubleshooting code by entering it one line at a time and finding out where it fails. It lets you see what libraries are available and explore those libraries.

Try typing more into the REPL to see what happens!

Returning to the serial console

When you're ready to leave the REPL and return to the serial console, simply press **Ctrl + D**. This will reload your board and reenter the serial console. You will restart the program you had running before entering the REPL. In the console window, you'll see any output from the program you had running. And if your program was affecting anything visual on the board, you'll see that start up again as well.

You can return to the REPL at any time!

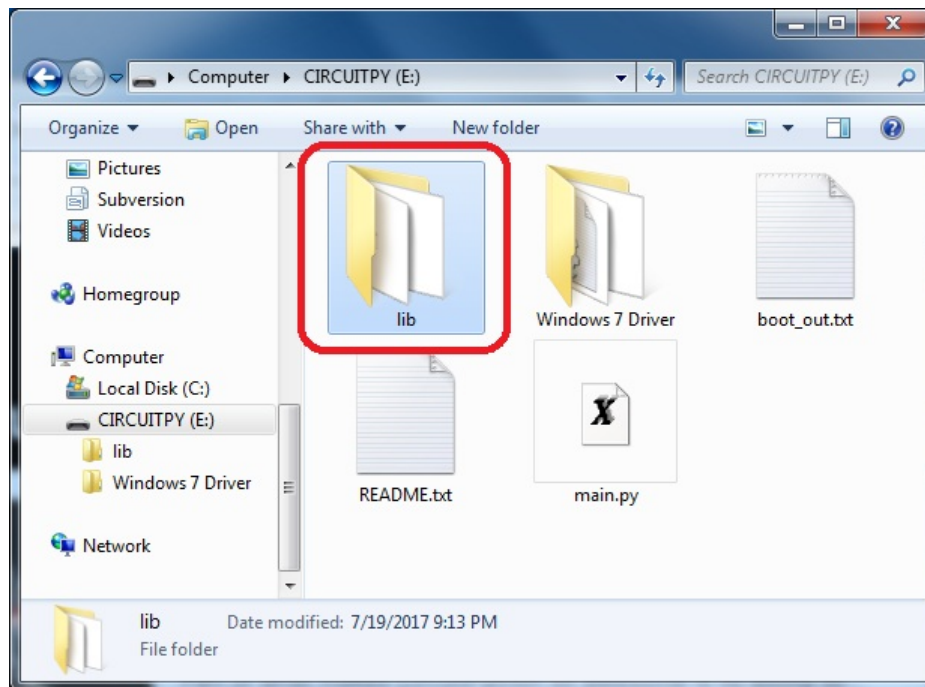
CircuitPython Libraries



As we continue to develop CircuitPython and create new releases, we will stop supporting older releases. If you are running CircuitPython 2.x or 3.x, you need to update to 4.x. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update to CircuitPython 4.x and then download the 4.x bundle.

Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. These files are called *libraries*. Some of them are built into CircuitPython. Others are stored on your **CIRCUITPY** drive in a folder called **lib**. Part of what makes CircuitPython so awesome is its ability to store code separately from the firmware itself. Storing code separately from the firmware makes it easier to update both the code you write and the libraries you depend.

Your board may ship with a **lib** folder already, it's in the base directory of the drive. If not, simply create the folder yourself.



CircuitPython libraries work in the same way as regular Python modules so the [Python docs \(https://adafru.it/rar\)](https://adafru.it/rar) are a great reference for how it all should work. In Python terms, we can place our library files in the **lib** directory because it's part of the Python path by default.

One downside of this approach of separate libraries is that they are not built in. To use them, one needs to copy them to the **CIRCUITPY** drive before they can be used. Fortunately, we provide a bundle full of our libraries.

Our bundle and releases also feature optimized versions of the libraries with the **.mpy** file extension. These files take less space on the drive and have a smaller memory footprint as they are loaded.

Installing the CircuitPython Library Bundle

We're constantly updating and improving our libraries, so we don't (at this time) ship our CircuitPython boards with the full library bundle. Instead, you can find example code in the guides for your board that depends on external libraries. Some of these libraries may be available from us at Adafruit, some may be written by community members!

Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.

You can grab the latest Adafruit CircuitPython Bundle release by clicking the button below.

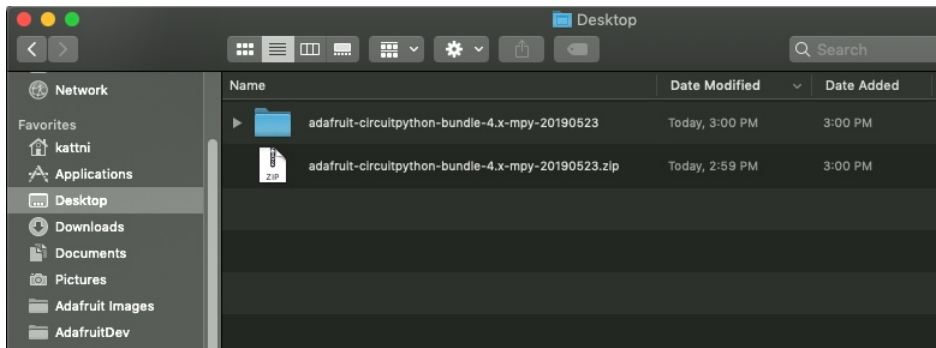
Note: Match up the bundle version with the version of CircuitPython you are running - 3.x library for running any version of CircuitPython 3, 4.x for running any version of CircuitPython 4, etc. If you mix libraries with major CircuitPython versions, you will most likely get errors due to changes in library interfaces possible during major version changes.

<https://adafru.it/ENC>
<https://adafru.it/ENC>

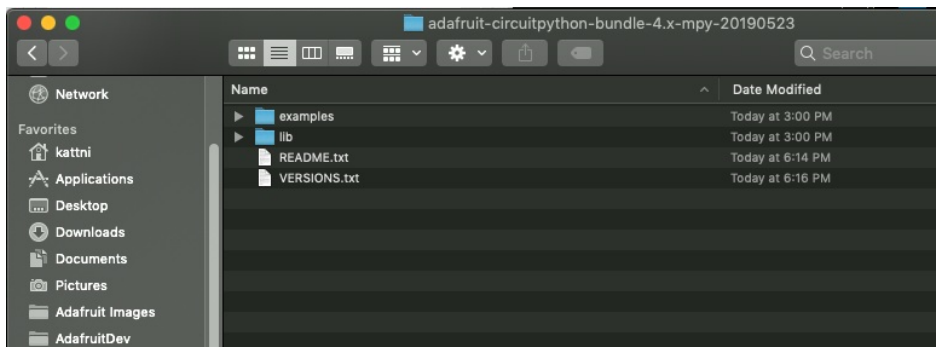
If you need another version, [you can also visit the bundle release page \(https://adafru.it/Ayy\)](https://adafru.it/Ayy) which will let you select exactly what version you're looking for, as well as information about changes.

Either way, download the version that matches your CircuitPython firmware version. If you don't know the version, look at the initial prompt in the CircuitPython REPL, which reports the version. For example, if you're running v4.0.1, download the 4.x library bundle. There's also a **py** bundle which contains the uncompressed python files, you probably *don't* want that unless you are doing advanced work on libraries.

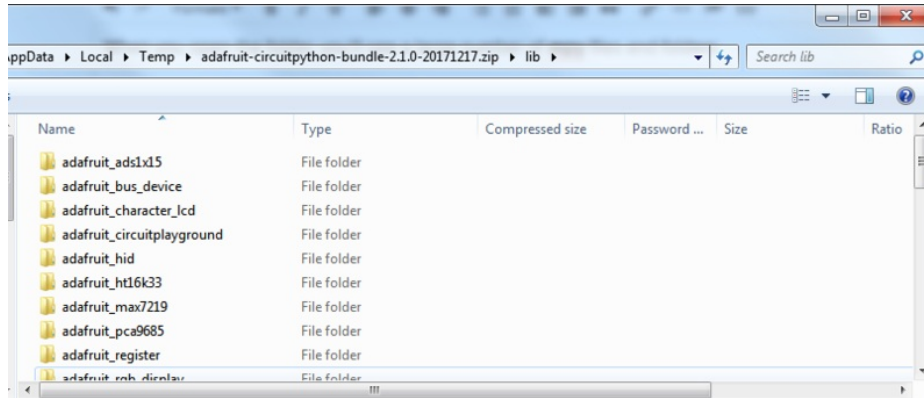
After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



Open the bundle folder. Inside you'll find two information files, and two folders. One folder is the lib bundle, and the other folder is the examples bundle.



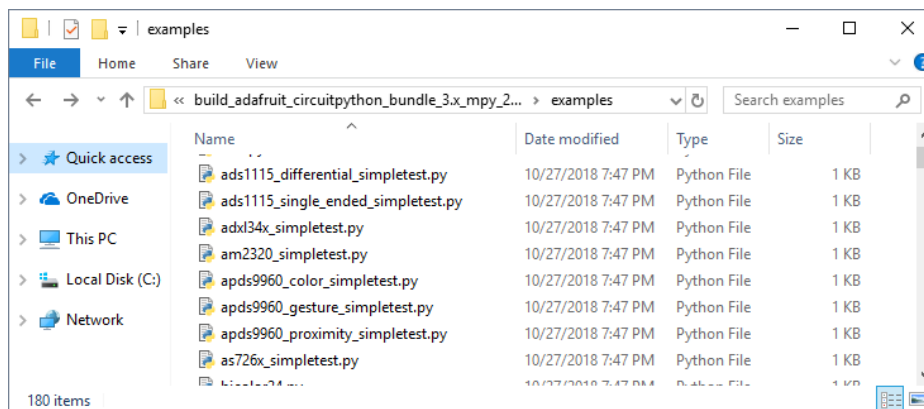
Now open the lib folder. When you open the folder, you'll see a large number of `mpy` files and folders



Example Files

All example files from each library are now included in the bundles, as well as an examples-only bundle. These are included for two main reasons:

- Allow for quick testing of devices.
- Provide an example base of code, that is easily built upon for individualized purposes.



Copying Libraries to Your Board

First you'll want to create a `lib` folder on your **CIRCUITPY** drive. Open the drive, right click, choose the option to create a new folder, and call it `lib`. Then, open the `lib` folder you extracted from the downloaded zip. Inside you'll find a number of folders and `.mpy` files. Find the library you'd like to use, and copy it to the `lib` folder on **CIRCUITPY**.

This also applies to example files. They are only supplied as raw `.py` files, so they may need to be converted to `.mpy` using the `mpy-cross` utility if you encounter `MemoryErrors`. This is discussed in the [CircuitPython Essentials Guide \(https://adafru.it/CTw\)](https://adafru.it/CTw). Usage is the same as described above in the Express Boards section. Note: If you do not place examples in a separate folder, you would remove the examples from the `import` statement.

Example: `ImportError` Due to Missing Library

If you choose to load libraries as you need them, you may write up code that tries to use a library you haven't yet loaded. We're going to demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

This demonstration will only return an error if you do not have the required library loaded into the `lib` folder on your **CIRCUITPY** drive.

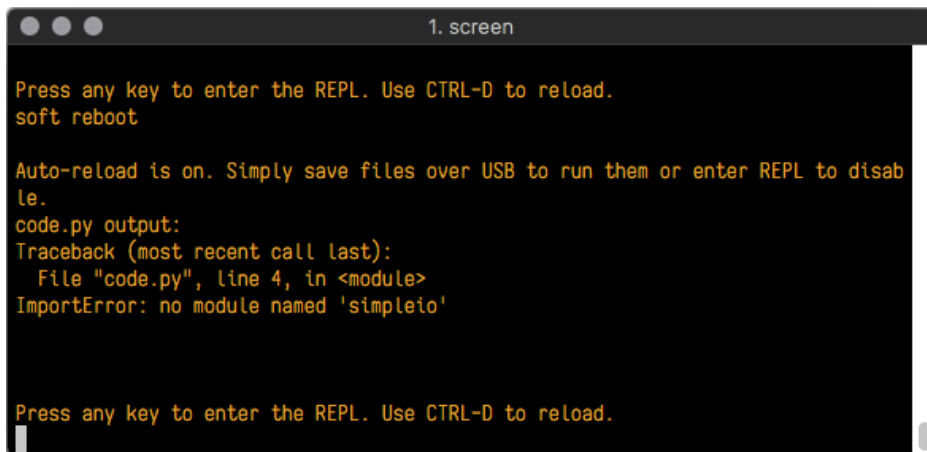
Let's use a modified version of the blinky example.

```
import board
import time
import simpleio

led = simpleio.DigitalOut(board.D13)

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.



```
1. screen

Press any key to enter the REPL. Use CTRL-D to reload.
soft reboot

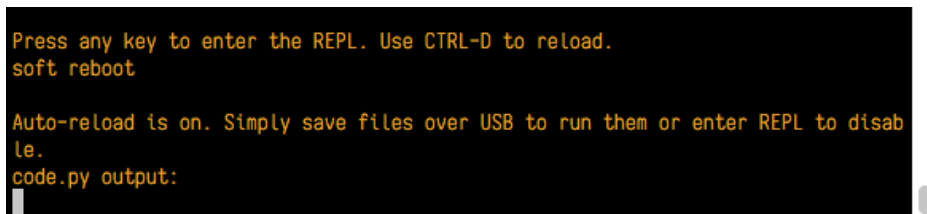
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Traceback (most recent call last):
  File "code.py", line 4, in <module>
    ImportError: no module named 'simpleio'

Press any key to enter the REPL. Use CTRL-D to reload.
```

We have an `ImportError`. It says there is `no module named 'simpleio'`. That's the one we just included in our code!

Click the link above to download the correct bundle. Extract the `lib` folder from the downloaded bundle file. Scroll down to find `simpleio.mpy`. This is the library file we're looking for! Follow the steps above to load an individual library file.

The LED starts blinking again! Let's check the serial console.



```
Press any key to enter the REPL. Use CTRL-D to reload.
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
```

No errors! Excellent. You've successfully resolved an `ImportError`!

If you run into this error in the future, follow along with the steps above and choose the library that matches the one you're missing.

Library Install on Non-Express Boards

If you have a Trinket M0 or Gemma M0, you'll want to follow the same steps in the example above to install libraries as you need them. You don't always need to wait for an `ImportError` as you probably know what library you added to your code. Simply open the `lib` folder you downloaded, find the library you need, and drag it to the `lib` folder on your **CIRCUITPY** drive.

You may end up running out of space on your Trinket M0 or Gemma M0 even if you only load libraries as you need them. There are a number of steps you can use to try to resolve this issue. You'll find them in the Troubleshooting page in the Learn guides for your board.

Updating CircuitPython Libraries/Examples

Libraries and examples are updated from time to time, and it's important to update the files you have on your **CIRCUITPY** drive.

To update a single library or example, follow the same steps above. When you drag the library file to your lib folder, it will ask if you want to replace it. Say yes. That's it!

A new library bundle is released every time there's an update to a library. Updates include things like bug fixes and new features. It's important to check in every so often to see if the libraries you're using have been updated.

Frequently Asked Questions

These are some of the common questions regarding CircuitPython and CircuitPython microcontrollers.

□ Is ESP8266 or ESP32 supported in CircuitPython? Why not?

We are dropping ESP8266 support as of 4.x - For more information please read about it here!

<https://learn.adafruit.com/welcome-to-circuitpython/circuitpython-for-esp8266>

□ How do I connect to the Internet with CircuitPython?

If you'd like to add WiFi support, check out our [guide on ESP32/ESP8266 as a co-processor](#).

□ Is there asyncio support in CircuitPython

We do not have asyncio support in CircuitPython at this time

□ My RGB NeoPixel/DotStar LED is blinking funny colors - what does it mean?

The status LED can tell you what's going on with your CircuitPython board. [Read more here for what the colors mean!](#)

What is a `MemoryError`?

Memory allocation errors happen when you're trying to store too much on the board. The CircuitPython microcontroller boards have a limited amount of memory available. You can have about 250 lines of code on the M0 Express boards. If you try to `import` too many libraries, a combination of large libraries, or run a program with too many lines of code, your code will fail to run and you will receive a `MemoryError` in the serial console (REPL).

What do I do when I encounter a `MemoryError`?

Try resetting your board. Each time you reset the board, it reallocates the memory. While this is unlikely to resolve your issue, it's a simple step and is worth trying.

Make sure you are using `.mpy` versions of libraries. All of the CircuitPython libraries are available in the bundle in a `.mpy` format which takes up less memory than `.py` format. Be sure that you're using [the latest library bundle \(https://adafru.it/uap\)](https://adafru.it/uap) for your version of CircuitPython.

If that does not resolve your issue, try shortening your code. Shorten comments, remove extraneous or unneeded code, or any other clean up you can do to shorten your code. If you're using a lot of functions, you could try moving those into a separate library, creating a `.mpy` of that library, and importing it into your code.

You can turn your entire file into a `.mpy` and `import` that into `code.py`. This means you will be unable to edit your code live on the board, but it can save you space.

Can the order of my `import` statements affect memory?

It can because the memory gets fragmented differently depending on allocation order and the size of objects. Loading `.mpy` files uses less memory so its recommended to do that for files you aren't editing.

How can I create my own `.mpy` files?

You can make your own `.mpy` versions of files with `mpy-cross`.

You can download the CircuitPython 2.x version of `mpy-cross` for your operating system from the [CircuitPython Releases page \(https://adafru.it/tBa\)](https://adafru.it/tBa) under the latest 2.x version.

You can build `mpy-cross` for CircuitPython 3.x by cloning the [CircuitPython GitHub repo \(https://adafru.it/tB7\)](https://adafru.it/tB7), and running `make` in the `circuitpython/mpy-cross/` directory. Then run `./mpy-cross path/to/foo.py` to create a `foo.mpy` in the same directory as the original file.

How do I check how much memory I have free?

```
import gc
gc.mem_free()
```

Will give you the number of bytes available for use.

Does CircuitPython support interrupts?

No. CircuitPython does not currently support interrupts. We do not have an estimated time for when they will be included.

Does Feather M0 support WINC1500?

No, WINC1500 will not fit into the M0 flash space.

Can AVR's such as ATmega328 or ATmega2560 run CircuitPython?

No.

Commonly Used Acronyms

CP or CPy = [CircuitPython \(https://adafru.it/cpy-welcome\)](https://adafru.it/cpy-welcome)

CPC = [Circuit Playground Classic \(https://adafru.it/ncE\)](https://adafru.it/ncE)

CPX = [Circuit Playground Express \(https://adafru.it/wpF\)](https://adafru.it/wpF)

Troubleshooting

From time to time, you will run into issues when working with CircuitPython. Here are a few things you may encounter and how to resolve them.



As we continue to develop CircuitPython and create new releases, we will stop supporting older releases. If you are running CircuitPython 2.x, you need to update to 3.x. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update to CircuitPython 3.x and then download the 3.x bundle.

Always Run the Latest Version of CircuitPython and Libraries

As we continue to develop CircuitPython and create new releases, we will stop supporting older releases. **If you are running CircuitPython 2.x, you need to [update to 3.x \(https://adafru.it/Amd\)](https://adafru.it/Amd).**

You need to download the CircuitPython Library Bundle that matches your version of CircuitPython. **Please update to CircuitPython 3.x and then [download the 3.x bundle \(https://adafru.it/ABU\)](https://adafru.it/ABU).**

We will soon stop providing the 2.x bundle as an automatically created download on the Adafruit CircuitPython Bundle repo. If you must continue to use 2.x, you can still download the 2.x version of `mpy-cross` from the 2.x release of CircuitPython on the CircuitPython repo and create your own 2.x compatible .mpy library files. **However, it is best to update to 3.x for both CircuitPython and the library bundle.**

CPLAYBOOT, TRINKETBOOT, FEATHERBOOT, or GEMMABOOT Drive Not Present

You may have a different board.

Only Adafruit Express boards and the Trinket M0 and Gemma M0 boards ship with the [UF2 bootloader \(https://adafru.it/zbX\)](https://adafru.it/zbX) installed. Feather M0 Basic, Feather M0 Adalogger, and similar boards use a regular Arduino-compatible bootloader, which does not show a `boardnameBOOT` drive.

MakeCode

If you are running a [MakeCode \(https://adafru.it/zbY\)](https://adafru.it/zbY) program on Circuit Playground Express, press the reset button just once to get the `CPLAYBOOT` drive to show up. Pressing it twice will not work.

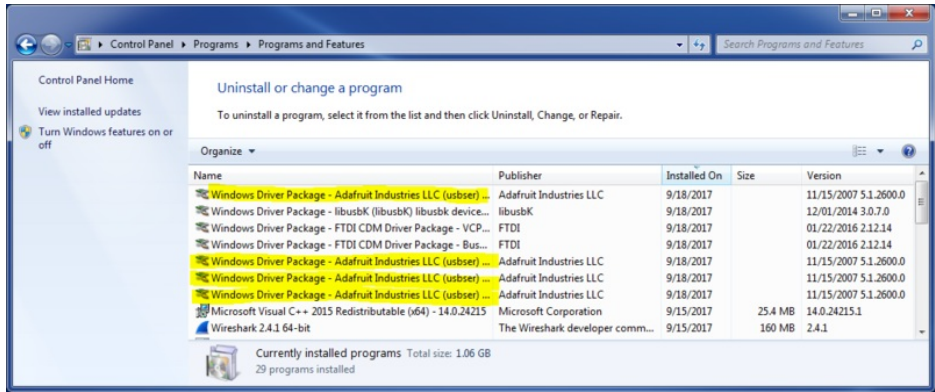
Windows 10

Did you install the Adafruit Windows Drivers package by mistake? You don't need to install this package on Windows 10 for most Adafruit boards. The old version (v1.5) can interfere with recognizing your device. Go to **Settings -> Apps** and uninstall all the "Adafruit" driver programs.

Windows 7

The latest version of the Adafruit Windows Drivers (version 2.0.0.0 or later) will fix the missing `boardnameBOOT` drive problem on Windows 7. To resolve this, first uninstall the old versions of the drivers:

- Unplug any boards. In **Uninstall or Change a Program (Control Panel->Programs->Uninstall a program)**, uninstall everything named "Windows Driver Package - Adafruit Industries LLC ...".

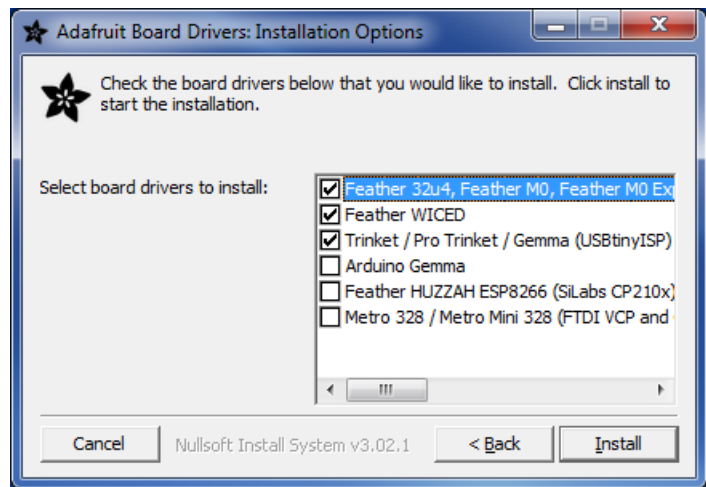


- Now install the new 2.3.0.0 (or higher) Adafruit Windows Drivers Package:

<https://adafru.it/ABO>

<https://adafru.it/ABO>

- When running the installer, you'll be shown a list of drivers to choose from. You can check and uncheck the boxes to choose which drivers to install.



You should now be done! Test by unplugging and replugging the board. You should see the **CIRCUITPY** drive, and when you double-click the reset button (single click on Circuit Playground Express running MakeCode), you should see the appropriate **boardnameBOOT** drive.

Let us know in the [Adafruit support forums \(https://adafru.it/jlf\)](https://adafru.it/jlf) or on the [Adafruit Discord \(\)](#) if this does not work for you!

Windows Explorer Locks Up When Accessing **boardnameBOOT** Drive

On Windows, several third-party programs we know of can cause issues. The symptom is that you try to access the **boardnameBOOT** drive, and Windows or Windows Explorer seems to lock up. These programs are known to cause trouble:

- **AIDA64**: to fix, stop the program. This problem has been reported to AIDA64. They acquired hardware to test, and released a beta version that fixes the problem. This may have been incorporated into the latest release.

Please let us know in the forums if you test this.

- **Hard Disk Sentinel**
- **Kaspersky anti-virus:** To fix, you may need to disable Kaspersky completely. Disabling some aspects of Kaspersky does not always solve the problem. This problem has been reported to Kaspersky.

CIRCUITPY Drive Does Not Appear

Kaspersky anti-virus can block the appearance of the **CIRCUITPY** drive. We haven't yet figured out a settings change that prevents this. Complete uninstallation of Kaspersky fixes the problem.

Norton anti-virus can interfere with **CIRCUITPY**. A user has reported this problem on Windows 7. The user turned off both Smart Firewall and Auto Protect, and CIRCUITPY then appeared.

Serial Console in Mu Not Displaying Anything

There are times when the serial console will accurately not display anything, such as, when no code is currently running, or when code with no serial output is already running before you open the console. However, if you find yourself in a situation where you feel it should be displaying something like an error, consider the following.

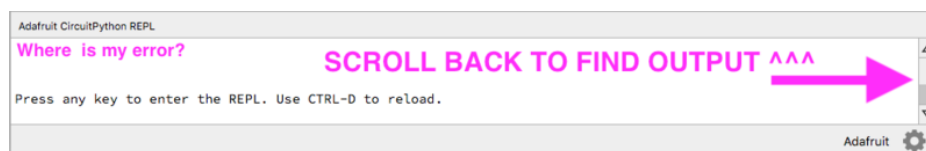
Depending on the size of your screen or Mu window, when you open the serial console, the serial console panel may be very small. This can be a problem. A basic CircuitPython error takes 10 lines to display!

```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Traceback (most recent call last):
  File "code.py", line 7
SyntaxError: invalid syntax
```

Press any key to enter the REPL. Use CTRL-D to reload.

More complex errors take even more lines!

Therefore, if your serial console panel is five lines tall or less, you may only see blank lines or blank lines followed by **Press any key to enter the REPL. Use CTRL-D to reload.** If this is the case, you need to either mouse over the top of the panel to utilise the option to resize the serial panel, or use the scrollbar on the right side to scroll up and find your message.



This applies to any kind of serial output whether it be error messages or print statements. So before you start trying to debug your problem on the hardware side, be sure to check that you haven't simply missed the serial messages due to serial output panel height.

CircuitPython RGB Status Light

The Feather M0 Express, Feather M4 Express, Metro M0 Express, Metro M4 Express, ItsyBitsy M0 Express, ItsyBitsy M4

Express, Gemma M0, and Trinket M0 all have a single NeoPixel or DotStar RGB LED on the board that indicates the status of CircuitPython.

Circuit Playground Express does NOT have a status LED. The LEDs will pulse green when in the bootloader. They do NOT indicate any status while running CircuitPython.

Here's what the colors and blinking mean:

- steady **GREEN**: `code.py` (or `code.txt`, `main.py`, or `main.txt`) is running
- pulsing **GREEN**: `code.py` (etc.) has finished or does not exist
- steady **YELLOW** at start up: (4.0.0-alpha.5 and newer) CircuitPython is waiting for a reset to indicate that it should start in safe mode
- pulsing **YELLOW**: Circuit Python is in safe mode: it crashed and restarted
- steady **WHITE**: REPL is running
- steady **BLUE**: `boot.py` is running

Colors with multiple flashes following indicate a Python exception and then indicate the line number of the error. The color of the first flash indicates the type of error:

- **GREEN**: IndentationError
- **CYAN**: SyntaxError
- **WHITE**: NameError
- **ORANGE**: OSError
- **PURPLE**: ValueError
- **YELLOW**: other error

These are followed by flashes indicating the line number, including place value. **WHITE** flashes are thousands' place, **BLUE** are hundreds' place, **YELLOW** are tens' place, and **CYAN** are one's place. So for example, an error on line 32 would flash **YELLOW** three times and then **CYAN** two times. Zeroes are indicated by an extra-long dark gap.

ValueError: Incompatible `.mpy` file.

This error occurs when importing a module that is stored as a `mpy` binary file that was generated by a different version of CircuitPython than the one its being loaded into. In particular, the `mpy` binary format changed between CircuitPython versions 2.x and 3.x, as well as between 1.x and 2.x.

So, for instance, if you upgraded to CircuitPython 3.x from 2.x you'll need to download a newer version of the library that triggered the error on `import`. They are all available in the [Adafruit bundle \(https://adafru.it/y8E\)](https://adafru.it/y8E).

Make sure to download a version with 2.0.0 or higher in the filename if you're using CircuitPython version 2.2.4, and the version with 3.0.0 or higher in the filename if you're using CircuitPython version 3.0.

CIRCUITPY Drive Issues

You may find that you can no longer save files to your `CIRCUITPY` drive. You may find that your `CIRCUITPY` stops showing up in your file explorer, or shows up as `NO_NAME`. These are indicators that your filesystem has issues.

First check - have you used Arduino to program your board? If so, CircuitPython is no longer able to provide the USB services. Reset the board so you get a `boardnameBOOT` drive rather than a `CIRCUITPY` drive, copy the latest version of CircuitPython (`.uf2`) back to the board, then Reset. This may restore `CIRCUITPY` functionality.

If still broken - When the **CIRCUITPY** disk is not safely ejected before being reset by the button or being disconnected from USB, it may corrupt the flash drive. It can happen on Windows, Mac or Linux.

In this situation, the board must be completely erased and CircuitPython must be reloaded onto the board.



You WILL lose everything on the board when you complete the following steps. If possible, make a copy of your code before continuing.

Easiest Way: Use `storage.erase_filesystem()`

Starting with version 2.3.0, CircuitPython includes a built-in function to erase and reformat the filesystem. If you have an older version of CircuitPython on your board, you can [update to the newest version \(https://adafruit.it/Amd\)](https://adafruit.it/Amd) to do this.

1. [Connect to the CircuitPython REPL \(https://adafruit.it/Bec\)](https://adafruit.it/Bec) using Mu or a terminal program.
2. Type:

```
>>> import storage
>>> storage.erase_filesystem()
```

CIRCUITPY will be erased and reformatted, and your board will restart. That's it!

Old Way: For the Circuit Playground Express, Feather M0 Express, and Metro M0 Express:

If you can't get to the REPL, or you're running a version of CircuitPython before 2.3.0, and you don't want to upgrade, you can do this.

1. Download the correct erase file:

<https://adafruit.it/AdI>

<https://adafruit.it/AdI>

<https://adafruit.it/AdJ>

<https://adafruit.it/AdJ>

<https://adafruit.it/EVK>

<https://adafruit.it/EVK>

<https://adafruit.it/AdK>

<https://adafruit.it/AdK>

<https://adafruit.it/EoM>

<https://adafruit.it/EoM>

<https://adafruit.it/DjD>

<https://adafru.it/DjD>

<https://adafru.it/DBA>

<https://adafru.it/DBA>

<https://adafru.it/Eca>

<https://adafru.it/Eca>

2. Double-click the reset button on the board to bring up the `boardnameBOOT` drive.
3. Drag the erase `.uf2` file to the `boardnameBOOT` drive.
4. The onboard NeoPixel will turn yellow or blue, indicating the erase has started.
5. After approximately 15 seconds, the mainboard NeoPixel will light up green. On the NeoTrellis M4 this is the first NeoPixel on the grid
6. Double-click the reset button on the board to bring up the `boardnameBOOT` drive.
7. Drag the appropriate latest release of CircuitPython (<https://adafru.it/Amd>) `.uf2` file to the `boardnameBOOT` drive.

It should reboot automatically and you should see `CIRCUITPY` in your file explorer again.

If the LED flashes red during step 5, it means the erase has failed. Repeat the steps starting with 2.

If you haven't already downloaded the latest release of CircuitPython for your board, check out the installation page (<https://adafru.it/Amd>). You'll also need to install your libraries and code!

Old Way: For Non-Express Boards with a UF2 bootloader (Gemma M0, Trinket M0):

If you can't get to the REPL, or you're running a version of CircuitPython before 2.3.0, and you don't want to upgrade, you can do this.

1. Download the erase file:

<https://adafru.it/AdL>

<https://adafru.it/AdL>

2. Double-click the reset button on the board to bring up the `boardnameBOOT` drive.
3. Drag the erase `.uf2` file to the `boardnameBOOT` drive.
4. The boot LED will start flashing again, and the `boardnameBOOT` drive will reappear.
5. Drag the appropriate latest release CircuitPython (<https://adafru.it/Amd>) `.uf2` file to the `boardnameBOOT` drive.

It should reboot automatically and you should see `CIRCUITPY` in your file explorer again.

If you haven't already downloaded the latest release of CircuitPython for your board, check out the installation page (<https://adafru.it/Amd>) You'll also need to install your libraries and code!

Old Way: For non-Express Boards without a UF2 bootloader (Feather M0 Basic Proto, Feather Adalogger, Arduino Zero):

If you are running a version of CircuitPython before 2.3.0, and you don't want to upgrade, or you can't get to the REPL, you can do this.

Just [follow these directions to reload CircuitPython using bossac](#) (<https://adafru.it/Bed>), which will erase and re-create **CIRCUITPY**.

Running Out of File Space on Non-Express Boards

The file system on the board is very tiny. (Smaller than an ancient floppy disk.) So, its likely you'll run out of space but don't panic! There are a couple ways to free up space.

The board ships with the Windows 7 serial driver too! Feel free to delete that if you don't need it or have already installed it. Its ~12KiB or so.

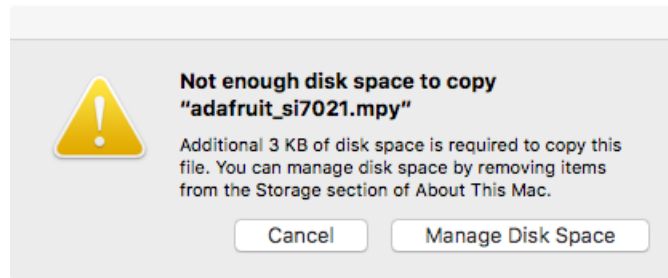
Delete something!

The simplest way of freeing up space is to delete files from the drive. Perhaps there are libraries in the **lib** folder that you aren't using anymore or test code that isn't in use.

Use tabs

One unique feature of Python is that the indentation of code matters. Usually the recommendation is to indent code with four spaces for every indent. In general, we recommend that too. **However**, one trick to storing more human-readable code is to use a single tab character for indentation. This approach uses 1/4 of the space for indentation and can be significant when we're counting bytes.

Mac OSX loves to add extra files.



Luckily you can disable some of the extra hidden files that Mac OSX adds by running a few commands to disable search indexing and create zero byte placeholders. Follow the steps below to maximize the amount of space available on OSX:

Prevent & Remove Mac OSX Hidden Files

First find the volume name for your board. With the board plugged in run this command in a terminal to list all the volumes:

```
ls -l /Volumes
```

Look for a volume with a name like **CIRCUITPY** (the default for CircuitPython). The full path to the volume is the **/Volumes/CIRCUITPY** path.

Now follow the [steps from this question](https://adafru.it/u1c) (<https://adafru.it/u1c>) to run these terminal commands that stop hidden files from being created on the board:

```
mdutil -i off /Volumes/CIRCUITPY
cd /Volumes/CIRCUITPY
rm -rf .{,_.}{fseventsd,Spotlight-V*,Trashes}
mkdir .fseventsd
touch .fseventsd/no_log .metadata_never_index .Trashes
cd -
```

Replace `/Volumes/CIRCUITPY` in the commands above with the full path to your board's volume if it's different. At this point all the hidden files should be cleared from the board and some hidden files will be prevented from being created.

However there are still some cases where hidden files will be created by Mac OSX. In particular if you copy a file that was downloaded from the internet it will have special metadata that Mac OSX stores as a hidden file. Luckily you can run a copy command from the terminal to copy files **without** this hidden metadata file. See the steps below.

Copy Files on Mac OSX Without Creating Hidden Files

Once you've disabled and removed hidden files with the above commands on Mac OSX you need to be careful to copy files to the board with a special command that prevents future hidden files from being created. Unfortunately you **cannot** use drag and drop copy in Finder because it will still create these hidden extended attribute files in some cases (for files downloaded from the internet, like Adafruit's modules).

To copy a file or folder use the `-X` option for the `cp` command in a terminal. For example to copy a `foo.mpy` file to the board use a command like:

```
cp -X foo.mpy /Volumes/CIRCUITPY
```

Or to copy a folder and all of its child files/folders use a command like:

```
cp -rX folder_to_copy /Volumes/CIRCUITPY
```

Other Mac OSX Space-Saving Tips

If you'd like to see the amount of space used on the drive and manually delete hidden files here's how to do so. First list the amount of space used on the `CIRCUITPY` drive with the `df` command:

Uninstalling CircuitPython

A lot of our boards can be used with multiple programming languages. For example, the Circuit Playground Express can be used with MakeCode, Code.org CS Discoveries, CircuitPython and Arduino.

Maybe you tried CircuitPython and want to go back to MakeCode or Arduino? Not a problem

You can always remove/re-install CircuitPython *whenever you want!* Heck, you can change your mind every day!

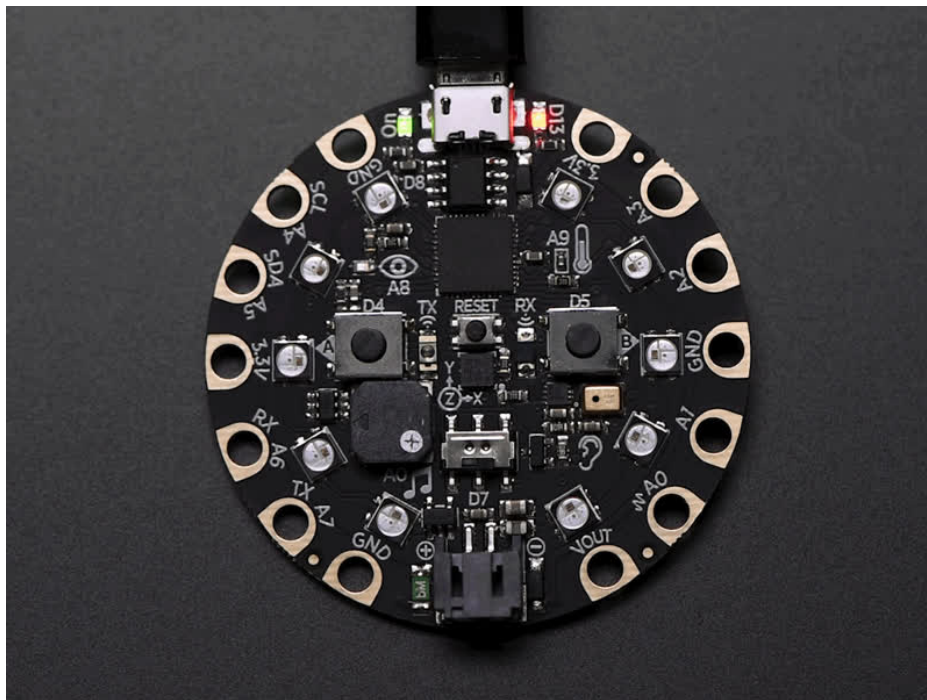
Backup Your Code

Before uninstalling CircuitPython, don't forget to make a backup of the code you have on the little disk drive. That means your **main.py** or **code.py** any other files, the **lib** folder etc. You may lose these files when you remove CircuitPython, so backups are key! Just drag the files to a folder on your laptop or desktop computer like you would with any USB drive.

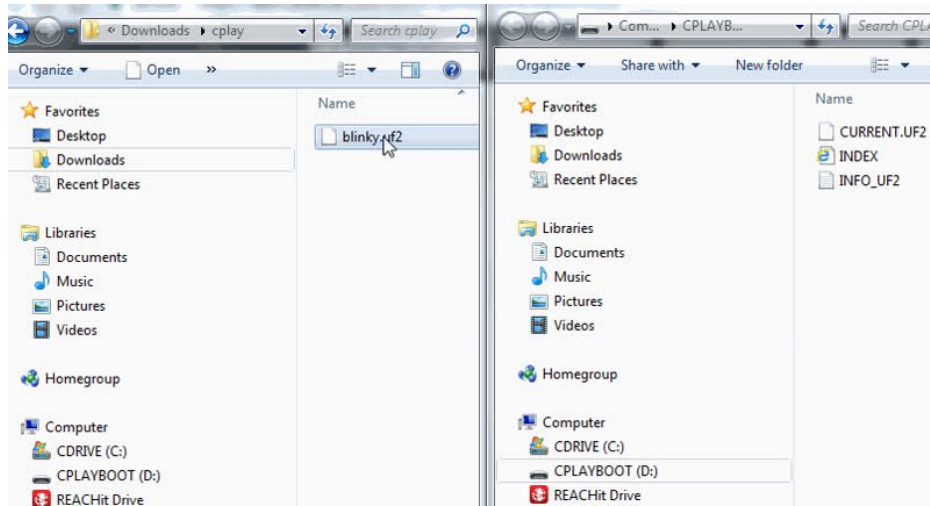
Moving to MakeCode

If you want to go back to using MakeCode, its really easy. Visit makecode.adafruit.com (<https://adafru.it/wpC>) and find the program you want to upload. Click Download to download the **.uf2** file that is generated by MakeCode.

Now double-click your CircuitPython board until you see the onboard LED(s) turn green and the **...BOOT** directory shows up.



Then find the downloaded MakeCode **.uf2** file and drag it to the **...BOOT** drive.



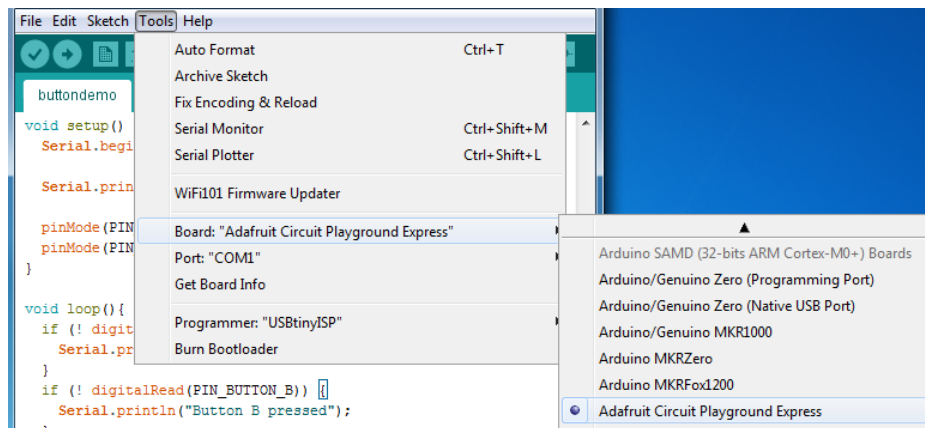
Your MakeCode is now running and CircuitPython has been removed. Going forward you only have to **single click** the reset button

Moving to Arduino

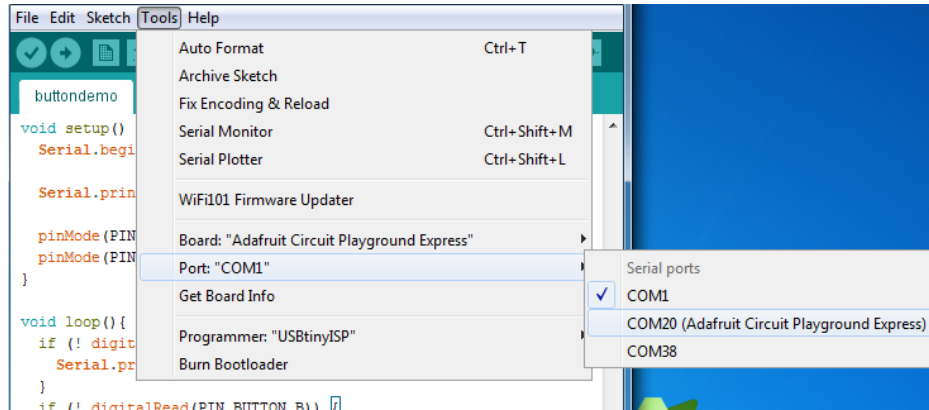
If you want to change your firmware to Arduino, it's also pretty easy.

Start by plugging in your board, and double-clicking reset until you get the green onboard LED(s) - just like with MakeCode

Within Arduino IDE, select the matching board, say Circuit Playground Express



Select the correct matching Port:



Create a new simple Blink sketch example:

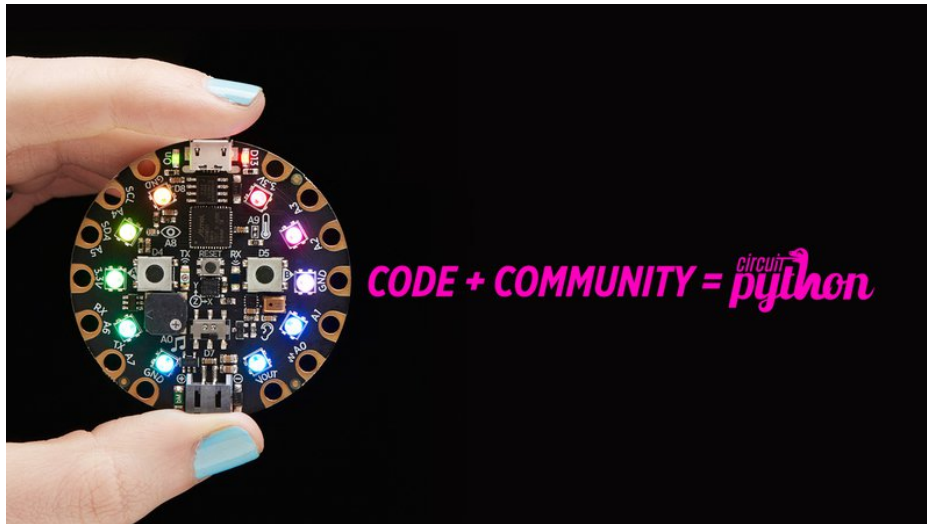
```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);           // wait for a second
  digitalWrite(13, LOW); // turn the LED off by making the voltage LOW
  delay(1000);          // wait for a second
}
```

Make sure the LED(s) are still green, then click **Upload** to upload Blink. Once it has uploaded successfully, the serial Port will change so **re-select the new Port!**

Once Blink is uploaded you should no longer need to double-click to enter bootloader mode, Arduino will automatically reset when you upload

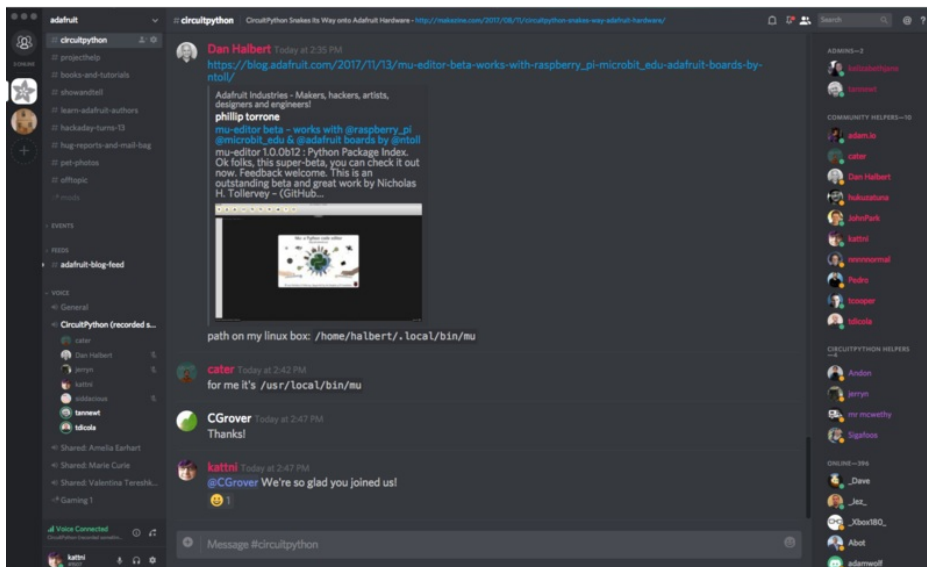
Welcome to the Community!



CircuitPython is a programming language that's super simple to get started with and great for learning. It runs on microcontrollers and works out of the box. You can plug it in and get started with any text editor. The best part? CircuitPython comes with an amazing, supportive community.

Everyone is welcome! CircuitPython is Open Source. This means it's available for anyone to use, edit, copy and improve upon. This also means CircuitPython becomes better because of you being a part of it. It doesn't matter whether this is your first microcontroller board or you're a computer engineer, you have something important to offer the Adafruit CircuitPython community. We're going to highlight some of the many ways you can be a part of it!

Adafruit Discord



The Adafruit Discord server is the best place to start. Discord is where the community comes together to volunteer and provide live support of all kinds. From general discussion to detailed problem solving, and everything in between, Discord is a digital maker space with makers from around the world.

There are many different channels so you can choose the one best suited to your needs. Each channel is shown on Discord as "#channelname". There's the #projecthelp channel for assistance with your current project or help coming up with ideas for your next one. There's the #showandtell channel for showing off your newest creation. Don't be afraid to ask a question in any channel! If you're unsure, #general is a great place to start. If another channel is more likely to provide you with a better answer, someone will guide you.

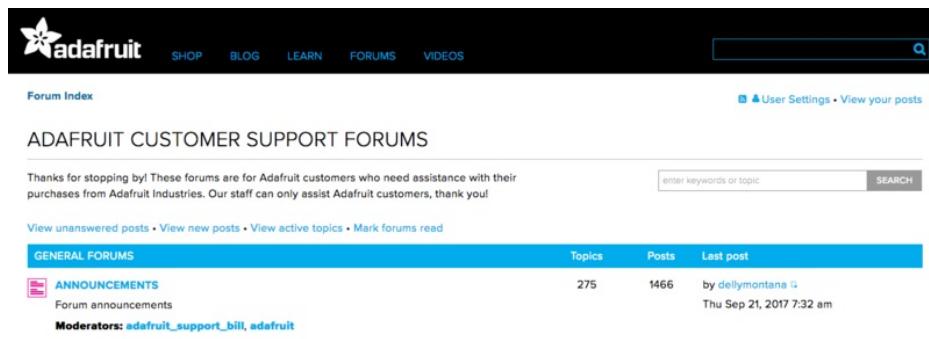
The CircuitPython channel is where to go with your CircuitPython questions. #circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. We'd love to hear what you have to say!

The easiest way to contribute to the community is to assist others on Discord. Supporting others doesn't always mean answering questions. Join in celebrating successes! Celebrate your mistakes! Sometimes just hearing that someone else has gone through a similar struggle can be enough to keep a maker moving forward.

The Adafruit Discord is the 24x7x365 hackerspace that you can bring your granddaughter to.

Visit <https://adafru.it/discord> ()to sign up for Discord. We're looking forward to meeting you!

Adafruit Forums




Forum Index User Settings • View your posts

ADAFRUIT CUSTOMER SUPPORT FORUMS

Thanks for stopping by! These forums are for Adafruit customers who need assistance with their purchases from Adafruit Industries. Our staff can only assist Adafruit customers, thank you!

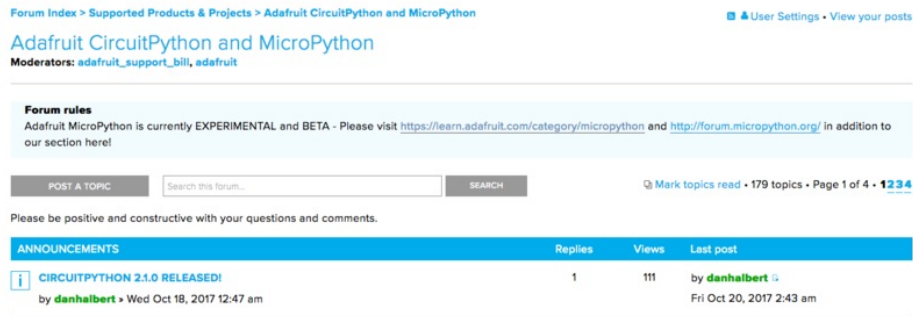
View unanswered posts • View new posts • View active topics • Mark forums read

GENERAL FORUMS	Topics	Posts	Last post
 ANNOUNCEMENTS Forum announcements Moderators: adafruit_support_bill , adafruit	275	1466	by dellymontana ↕ Thu Sep 21, 2017 7:32 am

The [Adafruit Forums \(https://adafru.it/jlf\)](https://adafru.it/jlf) are the perfect place for support. Adafruit has wonderful paid support folks to answer any questions you may have. Whether your hardware is giving you issues or your code doesn't seem to be working, the forums are always there for you to ask. You need an Adafruit account to post to the forums. You can use the same account you use to order from Adafruit.

While Discord may provide you with quicker responses than the forums, the forums are a more reliable source of information. If you want to be certain you're getting an Adafruit-supported answer, the forums are the best place to be.

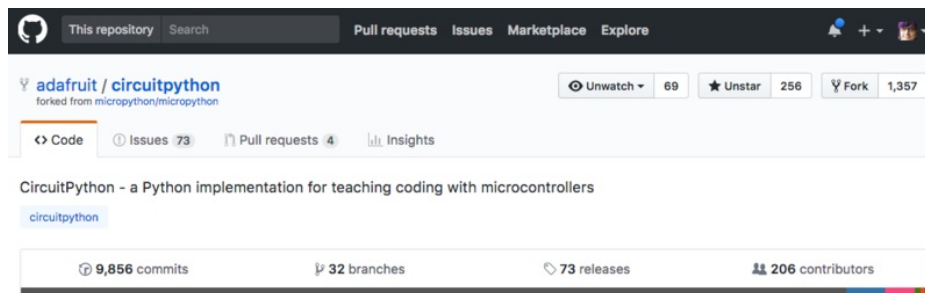
There are forum categories that cover all kinds of topics, including everything Adafruit. The [Adafruit CircuitPython and MicroPython \(https://adafru.it/xXA\)](https://adafru.it/xXA) category under "Supported Products & Projects" is the best place to post your CircuitPython questions.



Be sure to include the steps you took to get to where you are. If it involves wiring, post a picture! If your code is giving you trouble, include your code in your post! These are great ways to make sure that there's enough information to help you with your issue.

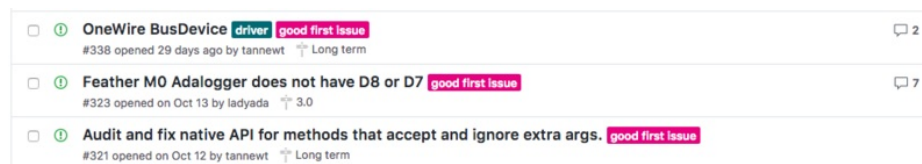
You might think you're just getting started, but you definitely know something that someone else doesn't. The great thing about the forums is that you can help others too! Everyone is welcome and encouraged to provide constructive feedback to any of the posted questions. This is an excellent way to contribute to the community and share your knowledge!

Adafruit Github



Whether you're just beginning or are life-long programmer who would like to contribute, there are ways for everyone to be a part of building CircuitPython. GitHub is the best source of ways to contribute to [CircuitPython \(https://adafru.it/tB7\)](https://adafru.it/tB7) itself. If you need an account, visit <https://github.com/> (<https://adafru.it/d6C>) and sign up.

If you're new to GitHub or programming in general, there are great opportunities for you. Head over to [adafruit/circuitpython \(https://adafru.it/tB7\)](https://adafru.it/tB7) on GitHub, click on "[Issues \(https://adafru.it/Bee\)](https://adafru.it/Bee)", and you'll find a list that includes issues labeled "[good first issue \(https://adafru.it/Bef\)](https://adafru.it/Bef)". These are things we've identified as something that someone with any level of experience can help with. These issues include options like updating documentation, providing feedback, and fixing simple bugs.



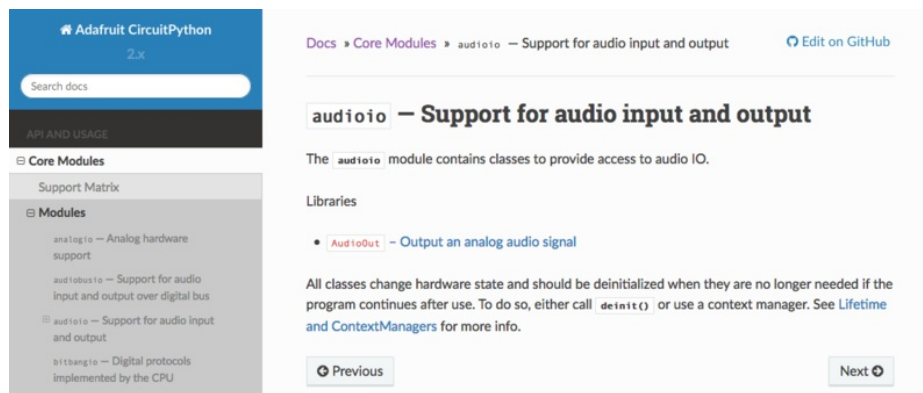
Already experienced and looking for a challenge? Checkout the rest of the issues list and you'll find plenty of ways to contribute. You'll find everything from new driver requests to core module updates. There's plenty of opportunities for everyone at any level!

When working with CircuitPython, you may find problems. If you find a bug, that's great! We love bugs! Posting a detailed issue to GitHub is an invaluable way to contribute to improving CircuitPython. Be sure to include the steps to replicate the issue as well as any other information you think is relevant. The more detail, the better!

Testing new software is easy and incredibly helpful. Simply load the newest version of CircuitPython or a library onto your CircuitPython hardware, and use it. Let us know about any problems you find by posting a new issue to GitHub. Software testing on both current and beta releases is a very important part of contributing CircuitPython. We can't possibly find all the problems ourselves! We need your help to make CircuitPython even better.

On GitHub, you can submit feature requests, provide feedback, report problems and much more. If you have questions, remember that Discord and the Forums are both there for help!

ReadTheDocs



[ReadTheDocs \(https://adafru.it/Beg\)](https://adafru.it/Beg) is an excellent resource for a more in depth look at CircuitPython. This is where you'll find things like API documentation and details about core modules. There is also a Design Guide that includes contribution guidelines for CircuitPython.

RTD gives you access to a low level look at CircuitPython. There are details about each of the [core modules \(https://adafru.it/Beh\)](https://adafru.it/Beh). Each module lists the available libraries. Each module library page lists the available parameters and an explanation for each. In many cases, you'll find quick code examples to help you understand how the modules and parameters work, however it won't have detailed explanations like the Learn Guides. If you want help understanding what's going on behind the scenes in any CircuitPython code you're writing, ReadTheDocs is there to help!

Here is blinky:

```
import digitalio
from board import *
import time

led = digitalio.DigitalInOut(D13)
led.direction = digitalio.Direction.OUTPUT
while True:
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.1)
```



You've gone through the [Welcome to CircuitPython guide \(https://adafru.it/cpy-welcome\)](https://adafru.it/cpy-welcome). You've already gotten everything setup, and you've gotten CircuitPython running. Great! Now what? CircuitPython Essentials!

There are a number of core modules built into CircuitPython and commonly used libraries available. This guide will introduce you to these and show you an example of how to use each one.

Each section will present you with a piece of code designed to work with different boards, and explain how to use the code with each board. These examples work with any board designed for CircuitPython, including **Circuit Playground Express**, **Trinket M0**, **Gemma M0**, **ItsyBitsy M0 Express**, **ItsyBitsy M4 Express**, **Feather M0 Express**, **Feather M4 Express**, **Metro M4 Express**, **Metro M0 Express**, **Trellis M4 Express**, and **Grand Central M4 Express**.

Some examples require external components, such as switches or sensors. You'll find wiring diagrams where applicable to show you how to wire up the necessary components to work with each example.

Let's get started learning the CircuitPython Essentials!

CircuitPython Built-Ins

CircuitPython comes 'with the kitchen sink' - a *lot* of the things you know and love about classic Python 3 (sometimes called CPython) already work. There are a few things that don't but we'll try to keep this list updated as we add more capabilities!

 This is not an exhaustive list! It's simply some of the many features you can use.

Thing That Are Built In and Work

Flow Control

All the usual `if`, `elif`, `else`, `for`, `while` work just as expected.

Math

`import math` will give you a range of handy mathematical functions.

```
>>> dir(math)
['__name__', 'e', 'pi', 'sqrt', 'pow', 'exp', 'log', 'cos', 'sin', 'tan', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'copysign', 'fabs', 'floor', 'fmod', 'frexp', 'ldexp', 'modf', 'isfinite', 'isinf', 'isnan', 'trunc', 'radians', 'degrees']
```

CircuitPython supports 30-bit wide floating point values so you can use `int` and `float` whenever you expect.

Tuples, Lists, Arrays, and Dictionaries

You can organize data in `()`, `[]`, and `{}` including strings, objects, floats, etc.

Classes, Objects and Functions

We use objects and functions extensively in our libraries so check out one of our many examples like this [MCP9808 library \(https://adafru.it/BfQ\)](https://adafru.it/BfQ) for class examples.

Lambdas

Yep! You can create function-functions with `lambda` just the way you like em:

```
>>> g = lambda x: x**2
>>> g(8)
64
```

Random Numbers

To obtain random numbers:

```
import random
```

`random.random()` will give a floating point number from `0` to `1.0`.

`random.randint(min, max)` will give you an integer number between `min` and `max`.

CircuitPython Digital In & Out

The first part of interfacing with hardware is being able to manage digital inputs and outputs. With CircuitPython, it's super easy!

This example shows how to use both a digital input and output. You can use a switch *input* with pullup resistor (built in) to control a digital *output* - the built in red LED.

Copy and paste the code into `code.py` using your favorite editor, and save the file to run the demo.

```
# CircuitPython IO demo #1 - General Purpose I/O
import time
import board
from digitalio import DigitalInOut, Direction, Pull

led = DigitalInOut(board.D13)
led.direction = Direction.OUTPUT

# For Gemma M0, Trinket M0, Metro M0 Express, ItsyBitsy M0 Express, Itsy M4 Express
switch = DigitalInOut(board.D2)
# switch = DigitalInOut(board.D5) # For Feather M0 Express, Feather M4 Express
# switch = DigitalInOut(board.D7) # For Circuit Playground Express
switch.direction = Direction.INPUT
switch.pull = Pull.UP

while True:
    # We could also do "led.value = not switch.value"!
    if switch.value:
        led.value = False
    else:
        led.value = True

    time.sleep(0.01) # debounce delay
```

Note that we made the code a little less "Pythonic" than necessary. The `if/else` block could be replaced with a simple `led.value = not switch.value` but we wanted to make it super clear how to test the inputs. The interpreter will read the digital input when it evaluates `switch.value`.

For **Gemma M0**, **Trinket M0**, **Metro M0 Express**, **Metro M4 Express**, **ItsyBitsy M0 Express**, **ItsyBitsy M4 Express**, no changes to the initial example are needed.



Note: To "comment out" a line, put a `#` and a space before it. To "uncomment" a line, remove the `#` + space from the beginning of the line.

For **Feather M0 Express** and **Feather M4 Express**, comment out `switch = DigitalInOut(board.D2)` (and/or `switch = DigitalInOut(board.D7)` depending on what changes you already made), and uncomment `switch = DigitalInOut(board.D5)`.

For **Circuit Playground Express**, you'll need to comment out `switch = DigitalInOut(board.D2)` (and/or `switch = DigitalInOut(board.D5)` depending on what changes you already made), and uncomment `switch =`

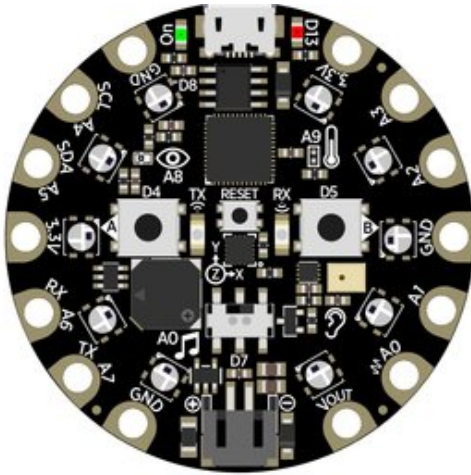
DigitalInOut(board.D7) .

To find the pin or pad suggested in the code, see the list below. For the boards that require wiring, wire up a switch (also known as a tactile switch, button or push-button), following the diagram for guidance. Press or slide the switch, and the onboard red LED will turn on and off.

Note that on the M0/SAMD based CircuitPython boards, at least, you can also have internal pulldowns with **Pull.DOWN** and if you want to turn off the pullup/pulldown just assign **switch.pull = None**.

Find the pins!

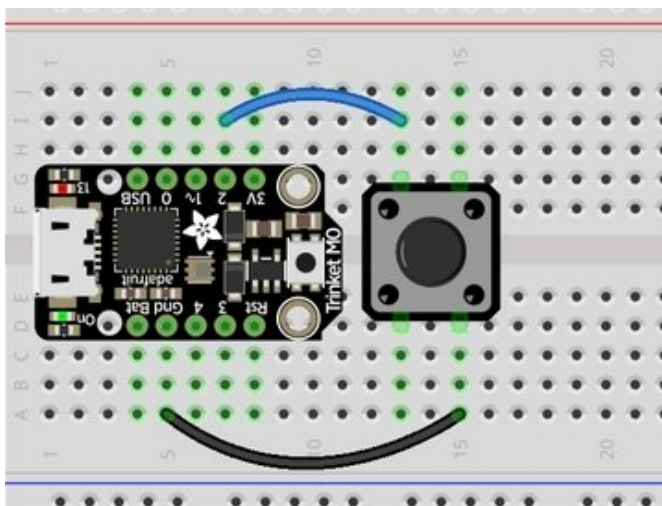
The list below shows each board, explains the location of the Digital pin suggested for use as input, and the location of the D13 LED.



Circuit Playground Express

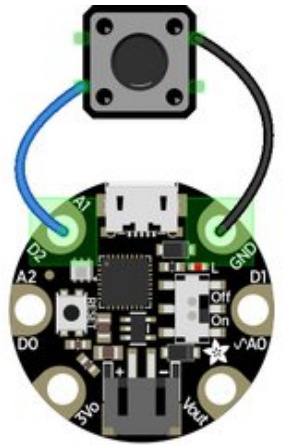
We're going to use the switch, which is pin D7, and is located between the battery connector and the reset switch on the board. D13 is labeled D13 and is located next to the USB micro port.

To use D7, comment out the current pin setup line, and uncomment the line labeled for Circuit Playground Express. See the details above!



Trinket M0

D2 is connected to the blue wire, labeled "2", and located between "3V" and "1" on the board. D13 is labeled "13" and is located next to the USB micro port.

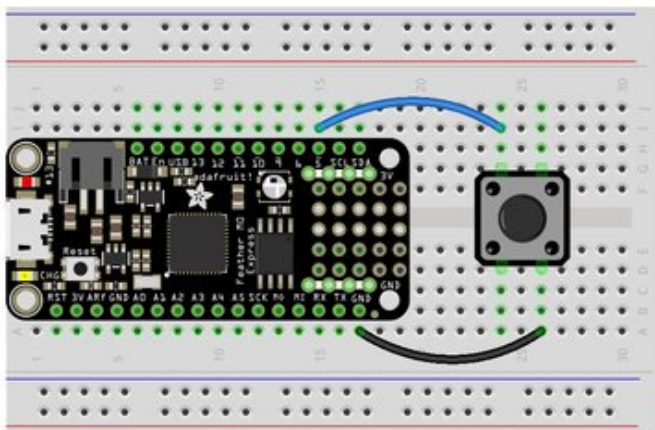


fritzing

Gemma M0

D2 is an alligator-clip-friendly pad labeled both "D2" and "A1", shown connected to the blue wire, and is next to the USB micro port. D13 is located next to the "GND" label on the board, above the "On/Off" switch.

Use alligator clips to connect your switch to your Gemma M0!

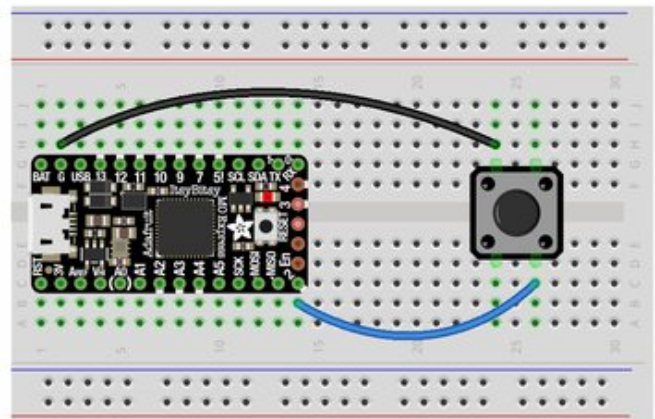


fritzing

Feather M0 Express and Feather M4 Express

D5 is labeled "5" and connected to the blue wire on the board. D13 is labeled "#13" and is located next to the USB micro port.

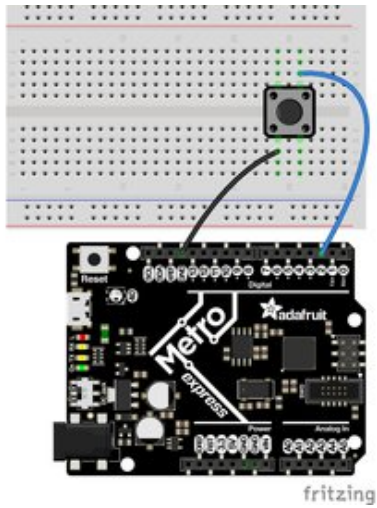
To use D5, comment out the current pin setup line, and uncomment the line labeled for Feather M0 Express. See the details above!



fritzing

ItsyBitsy M0 Express and ItsyBitsy M4 Express

D2 is labeled "2", located between the "MISO" and "EN" labels, and is connected to the blue wire on the board. D13 is located next to the reset button between the "3" and "4" labels on the board.



Metro M0 Express and Metro M4 Express

D2 is located near the top left corner, and is connected to the blue wire. D13 is labeled "L" and is located next to the USB micro port.

Read the Docs

For a more in-depth look at what [digitalio](#) can do, check out [the DigitalInOut page in Read the Docs](#) (<https://adafru.it/C4c>).

CircuitPython Analog In

This example shows you how you can read the analog voltage on the A1 pin on your board.

Copy and paste the code into `code.py` using your favorite editor, and save the file to run the demo.

```
# CircuitPython AnalogIn Demo
import time
import board
from analogio import AnalogIn

analog_in = AnalogIn(board.A1)

def get_voltage(pin):
    return (pin.value * 3.3) / 65536

while True:
    print((get_voltage(analog_in),))
    time.sleep(0.1)
```



Make sure you're running CircuitPython 3.x or higher! If you are running 2.x or lower, you may run into an error: "AttributeError: 'module' object has no attribute 'A1'". If you receive this error, first make sure you're running the latest version of CircuitPython!

Creating the analog input

```
analog_in = AnalogIn(board.A1)
```

Creates an object and connects the object to A1 as an analog input.

`get_voltage` Helper

`getVoltage(pin)` is our little helper program. By default, analog readings will range from 0 (minimum) to 65535 (maximum). This helper will convert the 0-65535 reading from `pin.value` and convert it to a 0-3.3V voltage reading.

Main Loop

The main loop is simple. It `prints` out the voltage as floating point values by calling `get_voltage` on our analog object. Connect to the serial console to see the results.



Changing It Up

By default the pins are *floating* so the voltages will vary. While connected to the serial console, try touching a wire from **A1** to the **GND** pin or **3Vo** pin to see the voltage change.

You can also add a potentiometer to control the voltage changes. From the potentiometer to the board, connect the **left pin** to **ground**, the **middle pin** to **A1**, and the **right pin** to **3V**. If you're using Mu editor, you can see the changes as you rotate the potentiometer on the plotter like in the image above! (Click the Plotter icon at the top of the window to open the plotter.)

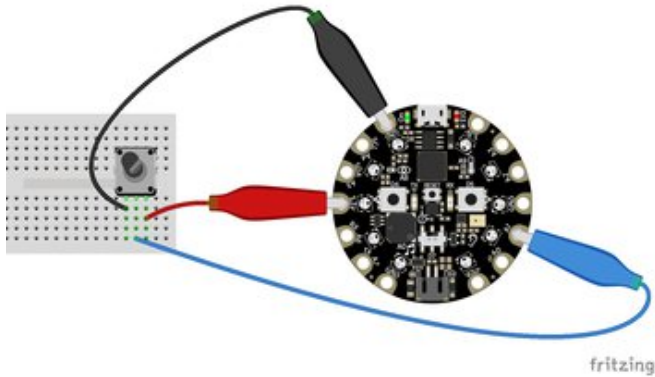


When you turn the knob of the potentiometer, the wiper rotates left and right, increasing or decreasing the resistance. This, in turn, changes the analog voltage level that will be read by your board on A1.

Wire it up

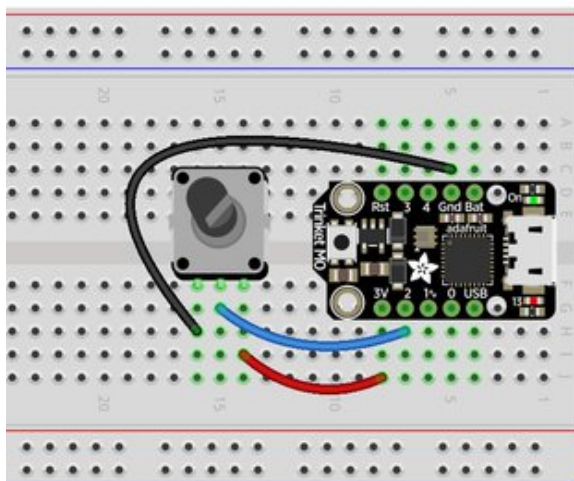
The list below shows wiring diagrams to help find the correct pins and wire up the potentiometer, and provides more information about analog pins on your board!

Circuit Playground Express



A1 is located on the right side of the board. There are multiple ground and 3V pads (pins).

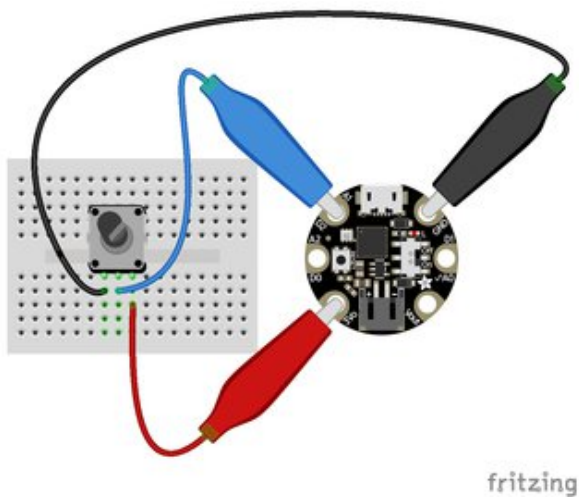
Your board has 7 analog pins that can be used for this purpose. For the full list, see the [pinout page \(https://adafru.it/AM9\)](https://adafru.it/AM9) on the main guide.



Trinket M0

A1 is labeled as 2! It's located between "1~" and "3V" on the same side of the board as the little red LED. Ground is located on the opposite side of the board. 3V is located next to 2, on the same end of the board as the reset button.

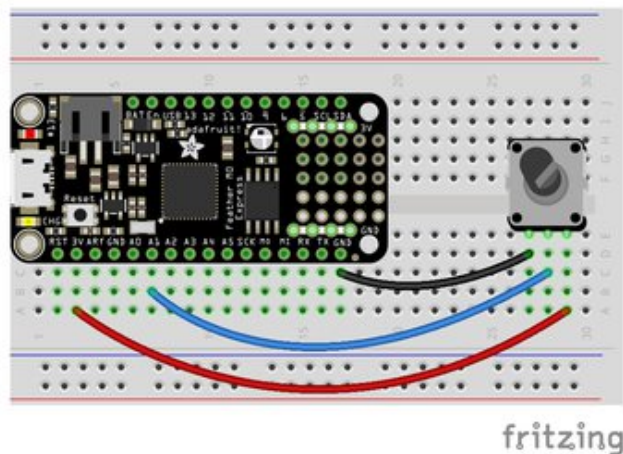
You have 5 analog pins you can use. For the full list, see the [pinouts page \(https://adafru.it/AMd\)](https://adafru.it/AMd) on the main guide.



Gemma M0

A1 is located near the top of the board of the board to the left side of the USB Micro port. Ground is on the other side of the USB port from A1. 3V is located to the left side of the battery connector on the bottom of the board.

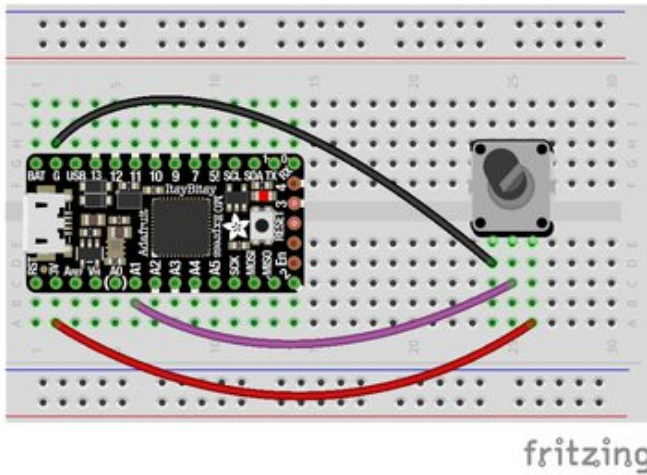
Your board has 3 analog pins. For the full list, see the [pinout page \(https://adafru.it/AMa\)](https://adafru.it/AMa) on the main guide.



Feather M0 Express and Feather M4 Express

A1 is located along the edge opposite the battery connector. There are multiple ground pins. 3V is located along the same edge as A1, and is next to the reset button.

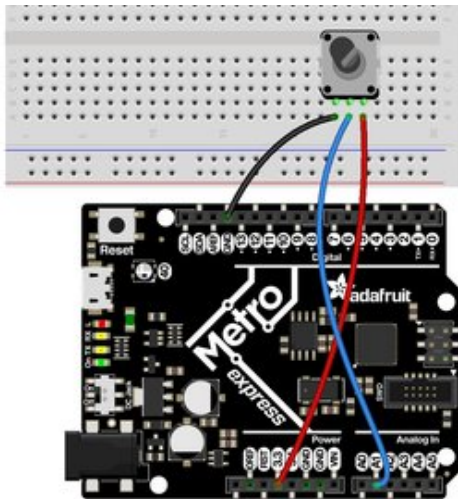
Your board has 6 analog pins you can use. For the full list, see the [pinouts page \(https://adafru.it/AMc\)](https://adafru.it/AMc) on the main guide.



ItsyBitsy M0 Express and ItsyBitsy M4 Express

A1 is located in the middle of the board, near the "A" in "Adafruit". Ground is labeled "G" and is located next to "BAT", near the USB Micro port. 3V is found on the opposite side of the USB port from Ground, next to RST.

You have 6 analog pins you can use. For a full list, see the [pinouts page \(https://adafru.it/BMg\)](https://adafru.it/BMg) on the main guide.



Metro M0 Express and Metro M4 Express

A1 is located on the same side of the board as the barrel jack. There are multiple ground pins available. 3V is labeled "3.3" and is located in the center of the board on the same side as the barrel jack (and as A1).

Your **Metro M0 Express** board has 6 analog pins you can use. For the full list, see the [pinouts page \(https://adafru.it/AMb\)](https://adafru.it/AMb) on the main guide.

Your **Metro M4 Express** board has 6 analog pins you can use. For the full list, see the [pinouts page \(https://adafru.it/B1O\)](https://adafru.it/B1O) on the main guide.

CircuitPython Analog Out

This example shows you how you can set the DAC (true analog output) on pin A0.

 A0 is the only true analog output on the M0 boards. No other pins do true analog output!

Copy and paste the code into `code.py` using your favorite editor, and save the file.

```
# CircuitPython IO demo - analog output
import board
from analogio import AnalogOut

analog_out = AnalogOut(board.A0)

while True:
    # Count up from 0 to 65535, with 64 increment
    # which ends up corresponding to the DAC's 10-bit range
    for i in range(0, 65535, 64):
        analog_out.value = i
```

Creating an analog output

```
analog_out = AnalogOut(A0)
```

Creates an object `analog_out` and connects the object to **A0**, the only DAC pin available on both the M0 and the M4 boards. (The M4 has two, A0 and A1.)

Setting the analog output

The DAC on the SAMD21 is a 10-bit output, from 0-3.3V. So in theory you will have a resolution of 0.0032 Volts per bit. To allow CircuitPython to be general-purpose enough that it can be used with chips with anything from 8 to 16-bit DACs, the DAC takes a 16-bit value and divides it down internally.

For example, writing 0 will be the same as setting it to 0 - 0 Volts out.

Writing 5000 is the same as setting it to $5000 / 64 = 78$, and $78 / 1024 * 3.3V = 0.25V$ output.

Writing 65535 is the same as 1023 which is the top range and you'll get 3.3V output

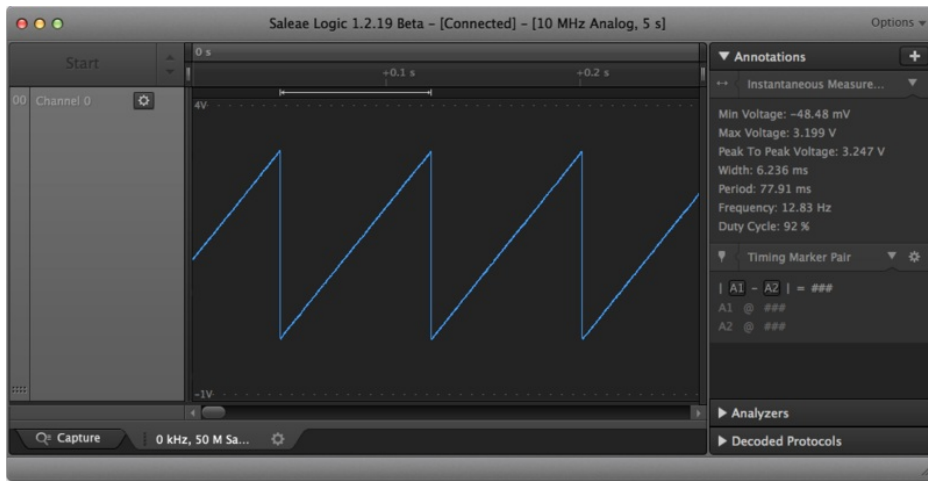
Main Loop

The main loop is fairly simple, it goes through the entire range of the DAC, from 0 to 65535, but increments 64 at a time so it ends up clicking up one bit for each of the 10-bits of range available.

CircuitPython is not terribly fast, so at the fastest update loop you'll get 4 Hz. The DAC isn't good for audio outputs as-is.

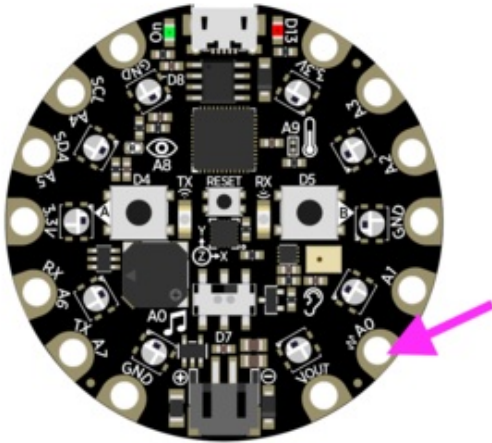
Express boards like the Circuit Playground Express, Metro M0 Express, ItsyBitsy M0 Express, ItsyBitsy M4 Express, Metro M4 Express, Feather M4 Express, or Feather M0 Express have more code space and can perform audio playback capabilities via the DAC. Gemma M0 and Trinket M0 cannot!

Check out [the Audio Out section of this guide \(https://adafru.it/BRj\)](https://adafru.it/BRj) for examples!



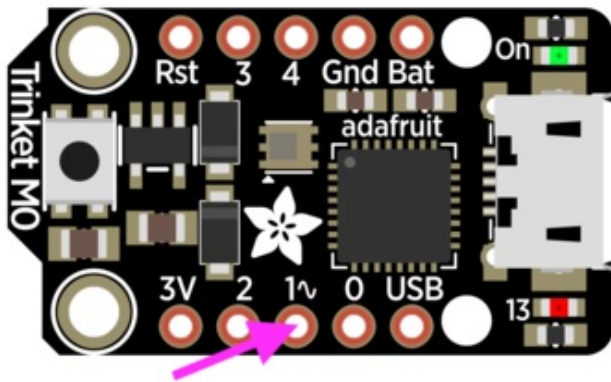
Find the pin

Use the diagrams below to find the A0 pin marked with a magenta arrow!



Circuit Playground Express

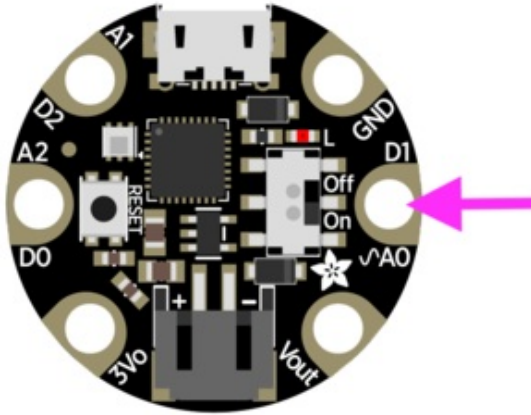
A0 is located between VOUT and A1 near the battery port.



Trinket M0

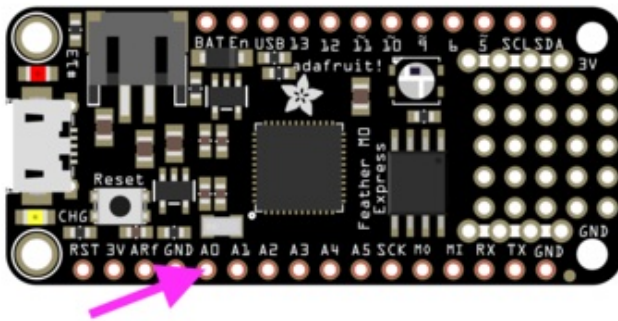
A0 is labeled "1~" on Trinket! A0 is located between "0" and "2" towards the middle of the board on the same side as the red LED.

Gemma M0



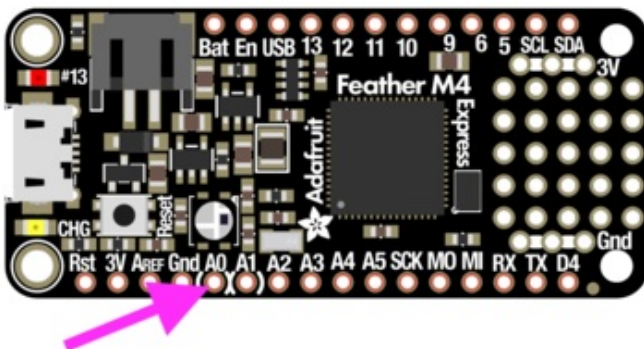
A0 is located in the middle of the right side of the board next to the On/Off switch.

Feather M0 Express



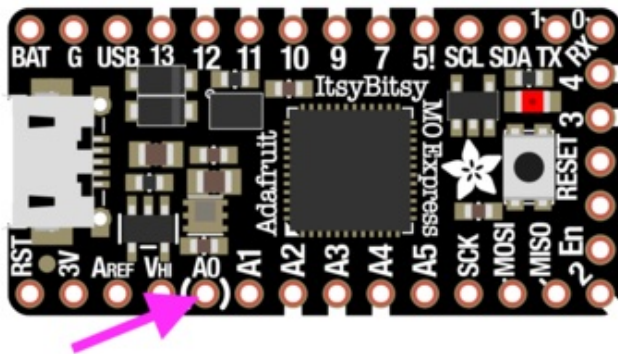
A0 is located between GND and A1 on the opposite side of the board from the battery connector, towards the end with the Reset button.

Feather M4 Express



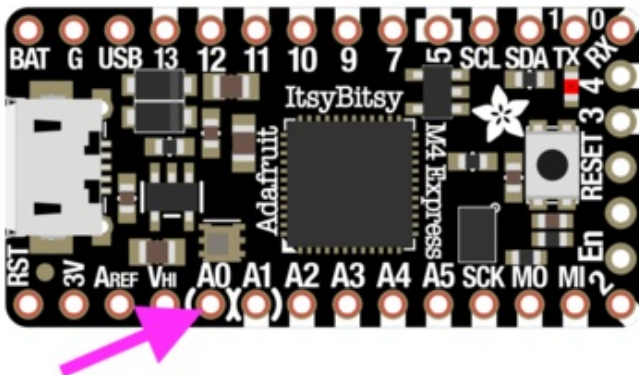
A0 is located between GND and A1 on the opposite side of the board from the battery connector, towards the end with the Reset button, and the pin pad has left and right white parenthesis markings around it

ItsyBitsy M0 Express



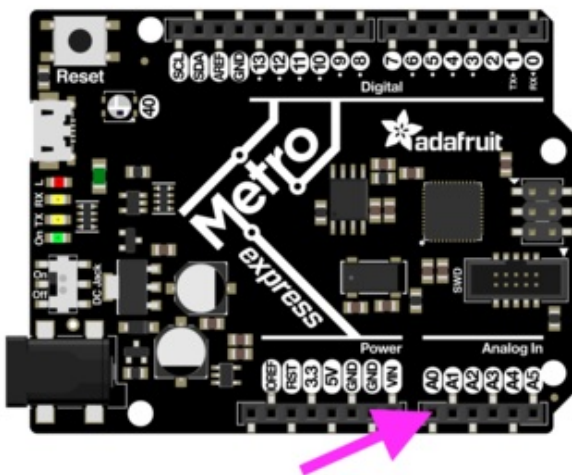
A0 is located between VHI and A1, near the "A" in "Adafruit", and the pin pad has left and right white parenthesis markings around it.

ItsyBitsy M4 Express

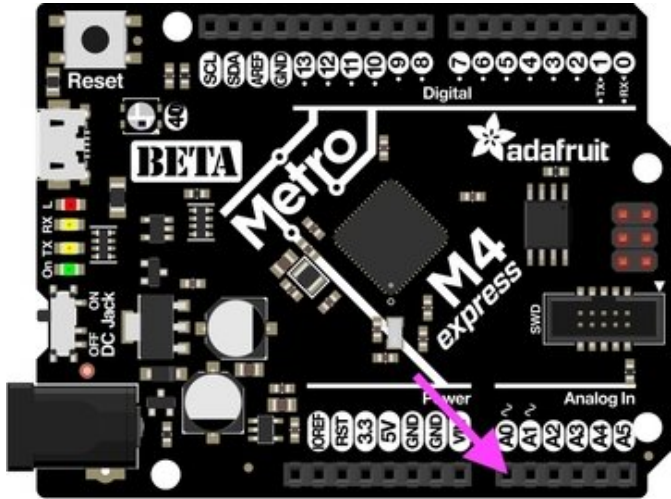


A0 is located between VHI and A1, and the pin pad has left and right white parenthesis markings around it.

Metro M0 Express



A0 is between VIN and A1, and is located along the same side of the board as the barrel jack adapter towards the middle of the headers found on that side of the board.



Metro M4 Express

A0 is between VIN and A1, and is located along the same side of the board as the barrel jack adapter towards the middle of the headers found on that side of the board.

On the Metro M4 Express, there are TWO true analog outputs: A0 and A1.

CircuitPython PWM

Your board has `pulseio` support, which means you can PWM LEDs, control servos, beep piezos, and manage "pulse train" type devices like DHT22 and Infrared.

Nearly every pin has PWM support! For example, all ATSAMD21 board have an **A0** pin which is 'true' analog out and *does not* have PWM support.

PWM with Fixed Frequency

This example will show you how to use PWM to fade the little red LED on your board.

Copy and paste the code into `code.py` using your favorite editor, and save the file.

```
import time
import board
import pulseio

led = pulseio.PWMOut(board.D13, frequency=5000, duty_cycle=0)

while True:
    for i in range(100):
        # PWM LED up and down
        if i < 50:
            led.duty_cycle = int(i * 2 * 65535 / 100) # Up
        else:
            led.duty_cycle = 65535 - int((i - 50) * 2 * 65535 / 100) # Down
        time.sleep(0.01)
```

Create a PWM Output

```
led = pulseio.PWMOut(board.D13, frequency=5000, duty_cycle=0)
```

Since we're using the onboard LED, we'll call the object `led`, use `pulseio.PWMOut` to create the output and pass in the `D13` LED pin to use.

Main Loop

The main loop uses `range()` to cycle through the loop. When the range is below 50, it PWMs the LED brightness up, and when the range is above 50, it PWMs the brightness down. This is how it fades the LED brighter and dimmer!

The `time.sleep()` is needed to allow the PWM process to occur over a period of time. Otherwise it happens too quickly for you to see!

PWM Output with Variable Frequency

Fixed frequency outputs are great for pulsing LEDs or controlling servos. But if you want to make some beeps with a piezo, you'll need to vary the frequency.

The following example uses `pulseio` to make a series of tones on a piezo.

To use with any of the M0 boards, no changes to the following code are needed.

To use with the Metro M4 Express, ItsyBitsy M4 Express or the Feather M4 Express, you must comment out the `piezo = pulseio.PWMOut(board.A2, duty_cycle=0, frequency=440, variable_frequency=True)` line and uncomment the `piezo = pulseio.PWMOut(board.A1, duty_cycle=0, frequency=440, variable_frequency=True)` line. A2 is not a supported PWM pin on the M4 boards!



Remember: To "comment out" a line, put a # and a space before it. To "uncomment" a line, remove the # + space from the beginning of the line.

```
import time
import board
import pulseio

# For the M0 boards:
piezo = pulseio.PWMOut(board.A2, duty_cycle=0, frequency=440, variable_frequency=True)

# For the M4 boards:
# piezo = pulseio.PWMOut(board.A1, duty_cycle=0, frequency=440, variable_frequency=True)

while True:
    for f in (262, 294, 330, 349, 392, 440, 494, 523):
        piezo.frequency = f
        piezo.duty_cycle = 65536 // 2 # On 50%
        time.sleep(0.25) # On for 1/4 second
        piezo.duty_cycle = 0 # Off
        time.sleep(0.05) # Pause between notes
    time.sleep(0.5)
```

If you have `simpleio` library loaded into your `/lib` folder on your board, we have a nice little helper that makes a tone for you on a piezo with a single command.

To use with any of the M0 boards, no changes to the following code are needed.

To use with the Metro M4 Express, ItsyBitsy M4 Express or the Feather M4 Express, you must comment out the `simpleio.tone(board.A2, f, 0.25)` line and uncomment the `simpleio.tone(board.A1, f, 0.25)` line. A2 is not a supported PWM pin on the M4 boards!

```
import time
import board
import simpleio

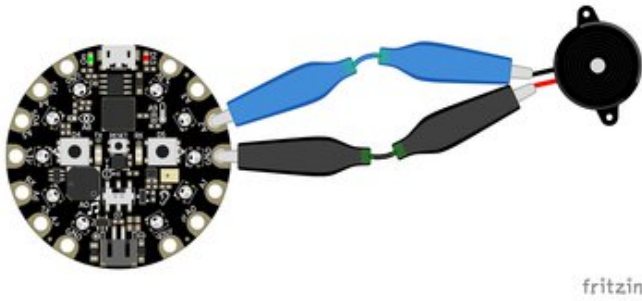
while True:
    for f in (262, 294, 330, 349, 392, 440, 494, 523):
        # For the M0 boards:
        simpleio.tone(board.A2, f, 0.25) # on for 1/4 second
        # For the M4 boards:
        # simpleio.tone(board.A1, f, 0.25) # on for 1/4 second
        time.sleep(0.05) # pause between notes
    time.sleep(0.5)
```

As you can see, it's much simpler!

Wire it up

Use the diagrams below to help you wire up your piezo. Attach one leg of the piezo to pin **A2** on the M0 boards or **A1** on the M4 boards, and the other leg to **ground**. It doesn't matter which leg is connected to which pin. They're interchangeable!

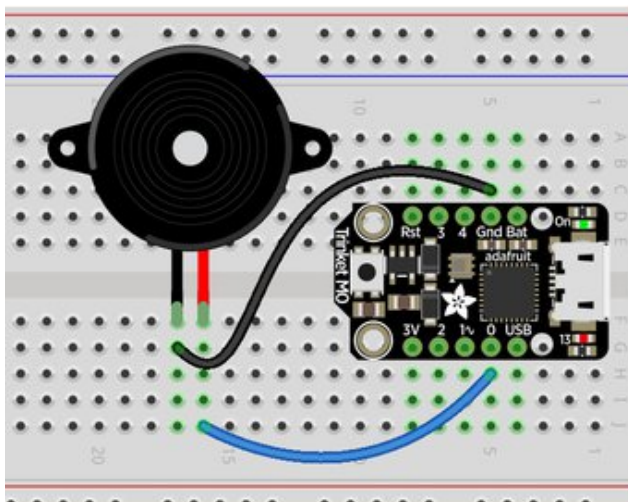
Circuit Playground Express



Use alligator clips to attach **A2** and any one of the **GND** to different legs of the piezo.

CPX has PWM on the following pins: A1, A2, A3, A6, RX, LIGHT, A8, TEMPERATURE, A9, BUTTON_B, D5, SLIDE_SWITCH, D7, D13, REMOTEIN, IR_RX, REMOTEOUT, IR_TX, IR_PROXIMITY, MICROPHONE_CLOCK, MICROPHONE_DATA, ACCELEROMETER_INTERRUPT, ACCELEROMETER_SDA, ACCELEROMETER_SCL, SPEAKER_ENABLE.

There is NO PWM on: A0, SPEAKER, A4, SCL, A5, SDA, A7, TX, BUTTON_A, D4, NEOPIXEL, D8, SCK, MOSI, MISO, FLASH_CS.



Trinket M0

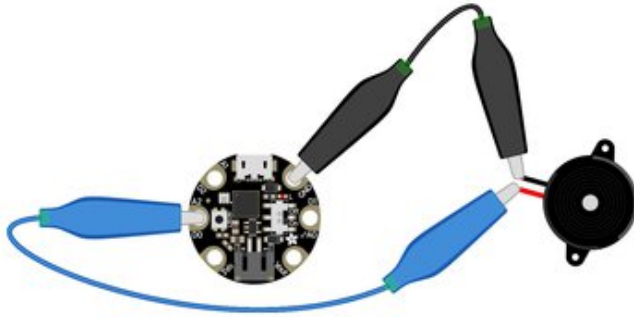
Note: A2 on Trinket is also labeled Digital "0"!

Use jumper wires to connect **GND** and **D0** to different legs of the piezo.

Trinket has PWM available on the following pins: D0, A2, SDA, D2, A1, SCL, MISO, D4, A4, TX, MOSI, D3, A3, RX, SCK, D13, APA102_MOSI, APA102_SCK.

There is NO PWM on: A0, D1.

Gemma M0



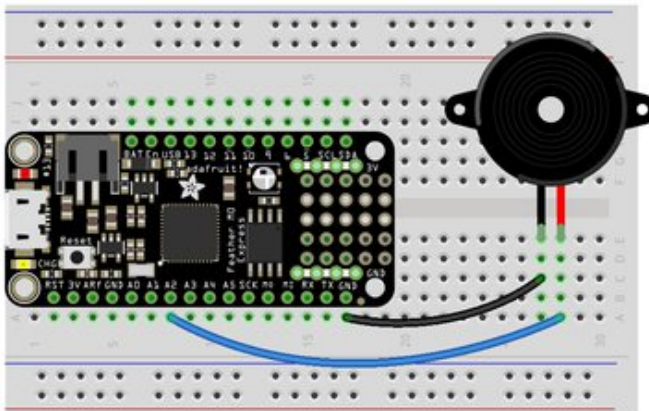
fritzing

Use alligator clips to attach **A2** and **GND** to different legs on the piezo.

Gemma has PWM available on the following pins: A1, D2, RX, SCL, A2, D0, TX, SDA, L, D13, APA102_MOSI, APA102_SCK.

There is NO PWM on: A0, D1.

Feather M0 Express



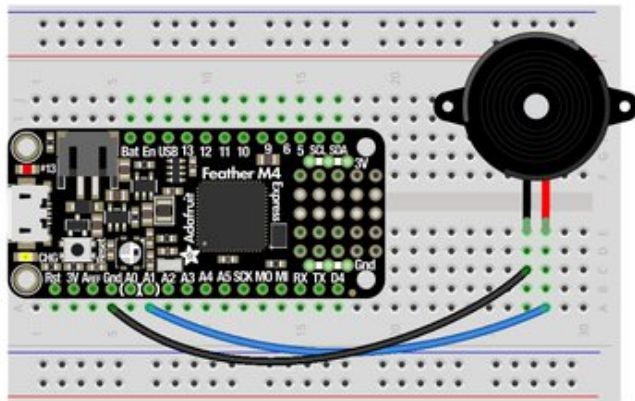
fritzing

Use jumper wires to attach **A2** and one of the two **GND** to different legs of the piezo.

Feather M0 Express has PWM on the following pins: A2, A3, A4, SCK, MOSI, MISO, D0, RX, D1, TX, SDA, SCL, D5, D6, D9, D10, D11, D12, D13, NEOPIXEL.

There is NO PWM on: A0, A1, A5.

Feather M4 Express



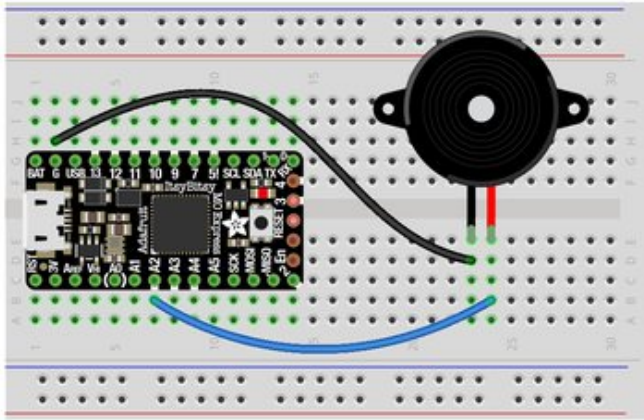
fritzing

Use jumper wires to attach **A1** and one of the two **GND** to different legs of the piezo.

To use A1, comment out the current pin setup line, and uncomment the line labeled for the M4 boards. See the details above!

Feather M4 Express has PWM on the following pins: A1, A3, SCK, D0, RX, D1, TX, SDA, SCL, D4, D5, D6, D9, D10, D11, D12, D13.

There is NO PWM on: A0, A2, A4, A5, MOSI, MISO.



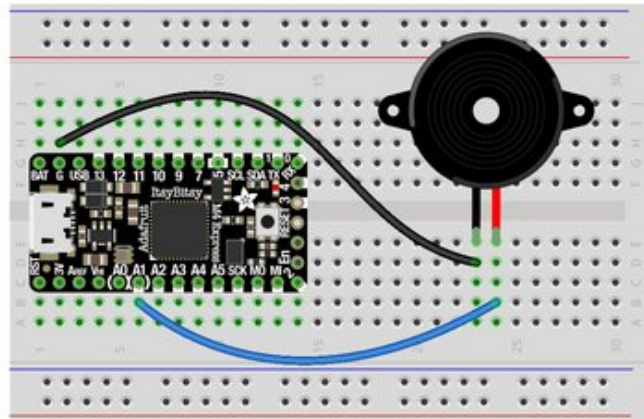
fritzing

ItsyBitsy M0 Express

Use jumper wires to attach **A2** and **G** to different legs of the piezo.

ItsyBitsy M0 Express has PWM on the following pins: D0, RX, D1, TX, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, L, A2, A3, A4, MOSI, MISO, SCK, SCL, SDA, APA102_MOSI, APA102_SCK.

There is NO PWM on: A0, A1, A5.



fritzing

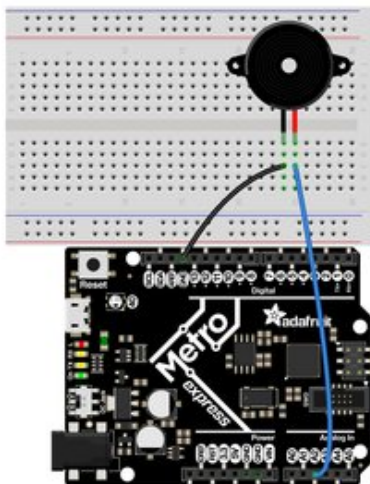
ItsyBitsy M4 Express

Use jumper wires to attach **A1** and **G** to different legs of the piezo.

To use A1, comment out the current pin setup line, and uncomment the line labeled for the M4 boards. See the details above!

ItsyBitsy M0 Express has PWM on the following pins: A1, D0, RX, D1, TX, D2, D4, D5, D7, D9, D10, D11, D12, D13, SDA, SCL.

There is NO PWM on: A2, A3, A4, A5, D3, SCK, MOSI, MISO.

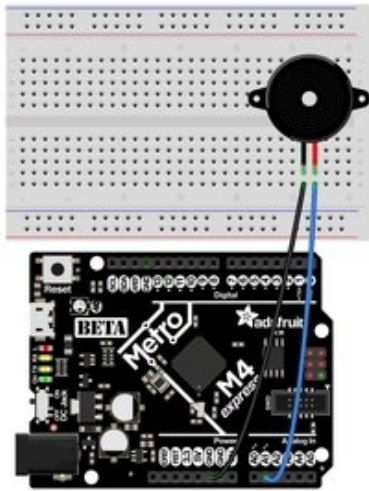


Metro M0 Express

Use jumper wires to connect **A2** and any one of the **GND** to different legs on the piezo.

Metro M0 Express has PWM on the following pins: A2, A3, A4, D0, RX, D1, TX, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, SDA, SCL, NEOPIXEL, SCK, MOSI, MISO.

There is NO PWM on: A0, A1, A5, FLASH_CS.



Metro M4 Express

Use jumper wires to connect **A1** and any one of the **GND** to different legs on the piezo.

To use A1, comment out the current pin setup line, and uncomment the line labeled for the M4 boards. See the details above!

Metro M4 Express has PWM on: A1, A5, D0, RX, D1, TX, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, SDA, SCK, MOSI, MISO

There is No PWM on: A0, A2, A3, A4, SCL, AREF, NEOPIXEL, LED_RX, LED_TX.

Where's My PWM?

Want to check to see which pins have PWM yourself? We've written this handy script! It attempts to setup PWM on every pin available, and lets you know which ones work and which ones don't. Check it out!

```
import board
import pulseio

for pin_name in dir(board):
    pin = getattr(board, pin_name)
    try:
        p = pulseio.PWMOut(pin)
        p.deinit()
        print("PWM on:", pin_name) # Prints the valid, PWM-capable pins!
    except ValueError: # This is the error returned when the pin is invalid.
        print("No PWM on:", pin_name) # Prints the invalid pins.
    except RuntimeError: # Timer conflict error.
        print("Timers in use:", pin_name) # Prints the timer conflict pins.
    except TypeError: # Error returned when checking a non-pin object in dir(board).
        pass # Passes over non-pin objects in dir(board).
```

CircuitPython Servo


In order to use servos, we take advantage of `pulseio`. Now, in theory, you could just use the raw `pulseio` calls to set the frequency to 50 Hz and then set the pulse widths. But we would rather make it a little more elegant and easy!

So, instead we will use `adafruit_motor` which manages servos for you quite nicely! `adafruit_motor` is a library so be sure to [grab it from the library bundle if you have not yet \(https://adafru.it/zdx\)](https://adafru.it/zdx)! If you need help installing the library, check out the [CircuitPython Libraries page \(https://adafru.it/ABU\)](https://adafru.it/ABU).

Servos come in two types:

- A **standard hobby servo** - the horn moves 180 degrees (90 degrees in each direction from zero degrees).
- A **continuous servo** - the horn moves 180 degrees like a DC motor. Instead of an angle specified, you set a throttle value with 1.0 being full forward, 0.5 being half forward, 0 being stopped, and -1 being full reverse, with other values between.

Servo Wiring

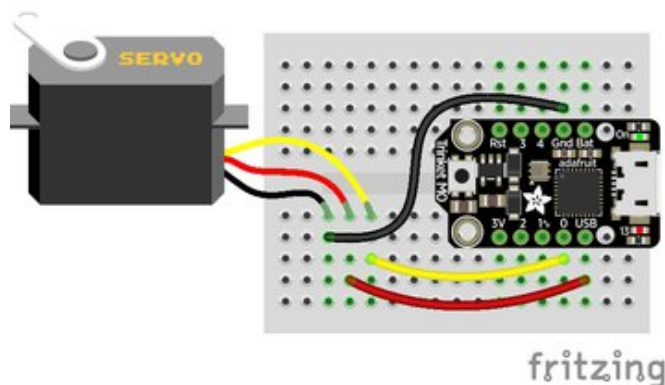
 Servos will only work on PWM-capable pins! Check your board details to verify which pins have PWM outputs.

The connections for a servo are the same for standard servos and continuous rotation servos.

Connect the servo's **brown** or **black** ground wire to ground on the CircuitPython board.

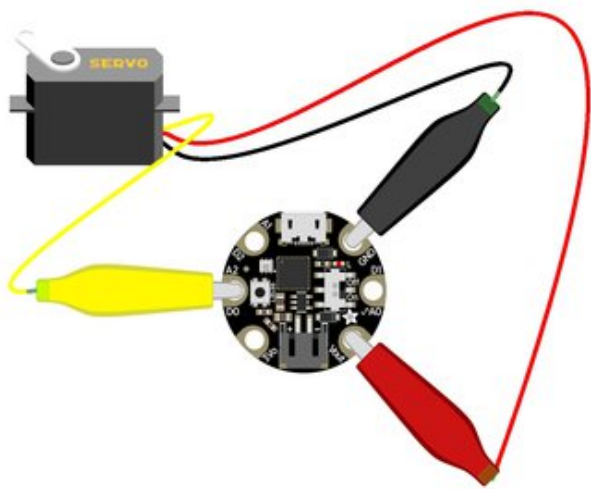
Connect the servo's **red** power wire to 5V power, USB power is good for a servo or two. For more than that, you'll need an external battery pack. Do not use 3.3V for powering a servo!

Connect the servo's **yellow** or **white** signal wire to the control/data pin, in this case **A1** or **A2** but you can use any PWM-capable pin.

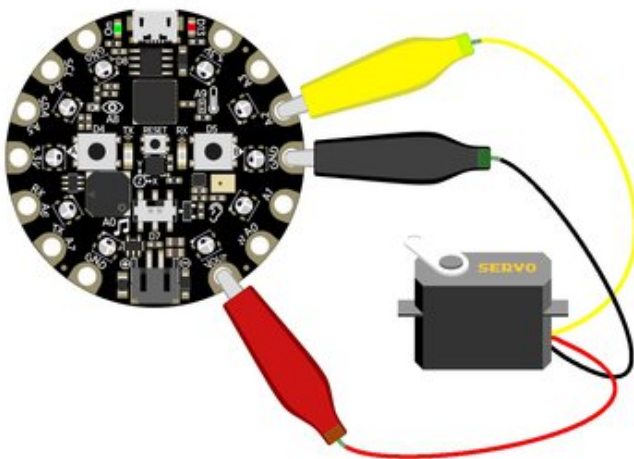


For example, to wire a servo to **Trinket**, connect the ground wire to **GND**, the power wire to **USB**, and the signal wire to **0**.

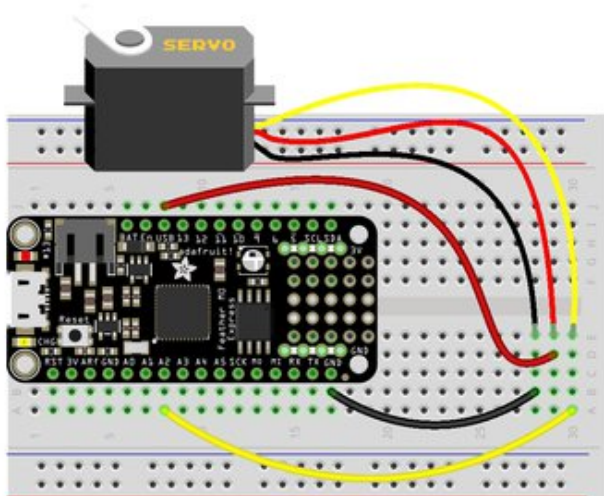
Remember, **A2** on Trinket is labeled "**0**".



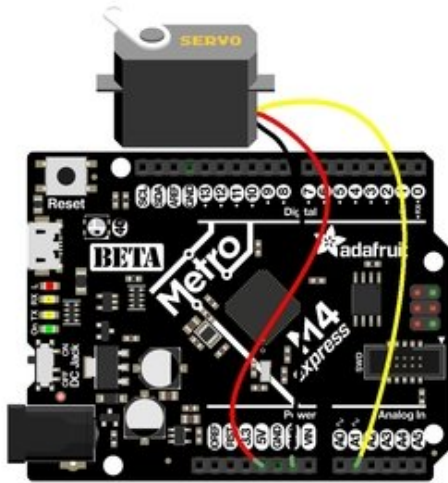
For **Gemma**, use jumper wire alligator clips to connect the ground wire to **GND**, the power wire to **VOUT**, and the signal wire to **A2**.



For **Circuit Playground Express**, use jumper wire alligator clips to connect the ground wire to **GND**, the power wire to **VOUT**, and the signal wire to **A2**.



For boards like **Feather M0 Express**, **ItsyBitsy M0 Express** and **Metro M0 Express**, connect the ground wire to any **GND**, the power wire to **USB or 5V**, and the signal wire to **A2**.



For the **Metro M4 Express**, **ItsyBitsy M4 Express** and the **Feather M4 Express**, connect the ground wire to any **G** or **GND**, the power wire to **USB** or **5V**, and the signal wire to **A1**.

Standard Servo Code

Here's an example that will sweep a servo connected to pin **A2** from 0 degrees to 180 degrees (-90 to 90 degrees) and back:

```
import time
import board
import pulseio
from adafruit_motor import servo

# create a PWMOut object on Pin A2.
pwm = pulseio.PWMOut(board.A2, duty_cycle=2 ** 15, frequency=50)

# Create a servo object, my_servo.
my_servo = servo.Servo(pwm)

while True:
    for angle in range(0, 180, 5): # 0 - 180 degrees, 5 degrees at a time.
        my_servo.angle = angle
        time.sleep(0.05)
    for angle in range(180, 0, -5): # 180 - 0 degrees, 5 degrees at a time.
        my_servo.angle = angle
        time.sleep(0.05)
```

Continuous Servo Code

There are two differences with Continuous Servos vs. Standard Servos:

1. The `servo` object is created like `my_servo = servo.ContinuousServo(pwm)` instead of `my_servo = servo.Servo(pwm)`
2. Instead of using `myservo.angle`, you use `my_servo.throttle` using a throttle value from 1.0 (full on) to 0.0 (stopped) to -1.0 (full reverse). Any number between would be a partial speed forward (positive) or reverse (negative). This is very similar to standard DC motor control with the `adafruit_motor` library.

This example runs full forward for 2 seconds, stops for 2 seconds, runs full reverse for 2 seconds, then stops for 4 seconds.

```

# Continuous Servo Test Program for CircuitPython
import time
import board
import pulseio
from adafruit_motor import servo

# create a PWMOut object on Pin A2.
pwm = pulseio.PWMOut(board.A2, frequency=50)

# Create a servo object, my_servo.
my_servo = servo.ContinuousServo(pwm)

while True:
    print("forward")
    my_servo.throttle = 1.0
    time.sleep(2.0)
    print("stop")
    my_servo.throttle = 0.0
    time.sleep(2.0)
    print("reverse")
    my_servo.throttle = -1.0
    time.sleep(2.0)
    print("stop")
    my_servo.throttle = 0.0
    time.sleep(4.0)

```

Pretty simple!

Note that we assume that 0 degrees is 0.5ms and 180 degrees is a pulse width of 2.5ms. That's a bit wider than the *official* 1-2ms pulse widths. If you have a servo that has a different range you can initialize the `servo` object with a different `min_pulse` and `max_pulse`. For example:

```
servo = adafruit_motor.servo.Servo(pwm, min_pulse = 0.5, max_pulse = 2.5)
```

For more detailed information on using servos with CircuitPython, check out the [CircuitPython section of the servo guide \(https://adafru.it/Bei\)](https://adafru.it/Bei)!

CircuitPython Cap Touch

Every CircuitPython designed M0 board has capacitive touch capabilities. This means each board has at least one pin that works as an input when you touch it! The capacitive touch is done *completely* in hardware, so no external resistors, capacitors or ICs required. Which is really nice!

 Capacitive touch is not supported on the M4 Express boards.

This example will show you how to use a capacitive touch pin on your board.

Copy and paste the code into **code.py** using your favorite editor, and save the file.

```
import time

import board
import touchio

touch_pad = board.A0 # Will not work for Circuit Playground Express!
# touch_pad = board.A1 # For Circuit Playground Express

touch = touchio.TouchIn(touch_pad)

while True:
    if touch.value:
        print("Touched!")
        time.sleep(0.05)
```


Create the Touch Input

First, we assign the variable `touch_pad` to a pin. The example uses **A0**, so we assign `touch_pad = board.A0`. You can choose any touch capable pin from the list below if you'd like to use a different pin. Then we create the touch object, name it `touch` and attach it to `touch_pad`.

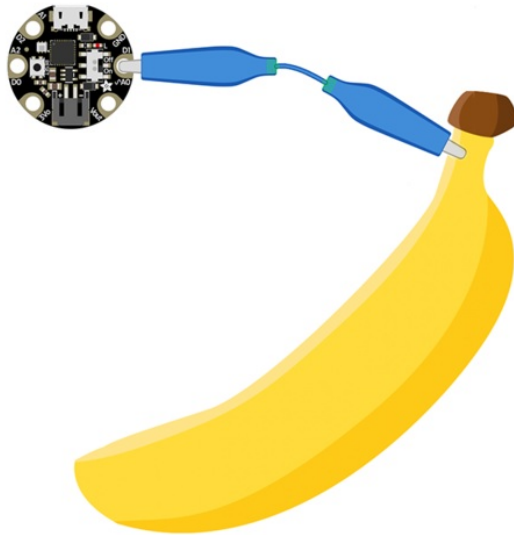
To use with Circuit Playground Express, comment out `touch_pad = board.A0` and uncomment `touch_pad = board.A1`.

Main Loop

Next, we create a loop that checks to see if the pin is touched. If it is, it `prints` to the serial console. Connect to the serial console to see the printed results when you touch the pin!

 Remember: To "comment out" a line, put a # and a space before it. To "uncomment" a line, remove the # + space from the beginning of the line.

No extra hardware is required, because you can touch the pin directly. However, you may want to attach alligator clips or copper tape to metallic or conductive objects. Try metal flatware, fruit or other foods, liquids, aluminum foil, or other items lying around your desk!



You may need to reload your code or restart your board after changing the attached item because the capacitive touch code "calibrates" based on what it sees when it first starts up. So if you get too many touch responses or not enough, reload your code through the serial console or eject the board and tap the reset button!

Find the Pin(s)

Your board may have more touch capable pins beyond A0. We've included a list below that helps you find A0 (or A1 in the case of CPX) for this example, identified by the magenta arrow. This list also includes information about any other pins that work for touch on each board!

To use the other pins, simply change the number in **A0** to the pin you want to use. For example, if you want to use **A3** instead, your code would start with `touch_pad = board.A3`.

If you would like to use more than one pin at the same time, your code may look like the following. If needed, you can modify this code to include pins that work for your board.

```
# CircuitPython Demo - Cap Touch Multiple Pins
# Example does NOT work with Trinket M0!

import time

import board
import touchio

touch_A1 = touchio.TouchIn(board.A1) # Not a touch pin on Trinket M0!
touch_A2 = touchio.TouchIn(board.A2) # Not a touch pin on Trinket M0!

while True:
    if touch_A1.value:
        print("Touched A1!")
    if touch_A2.value:
        print("Touched A2!")
    time.sleep(0.05)
```

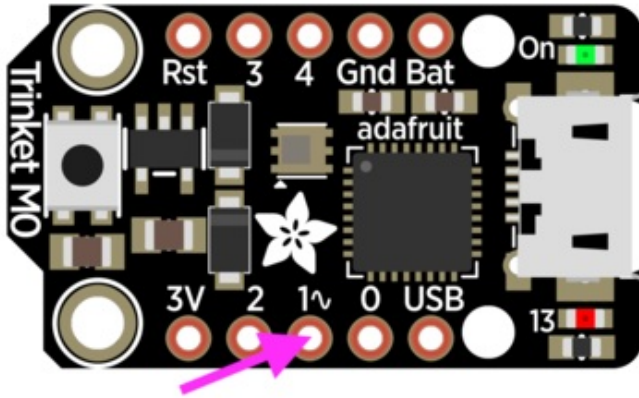
This example does NOT work for Trinket M0! You must change the pins to use with this board. This example



only works with Gemma, Circuit Playground Express, Feather M0 Express, Metro M0 Express and ItsyBitsy M0 Express.

Use the list below to find out what pins you can use with your board. Then, try adding them to your code and have fun!

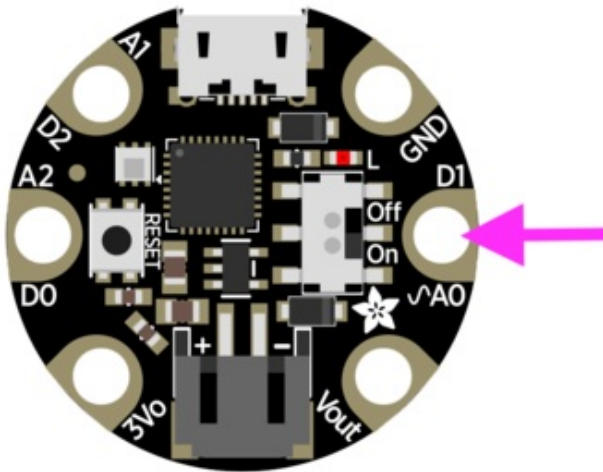
Trinket M0



There are three touch capable pins on Trinket: **A0**, **A3**, and **A4**.

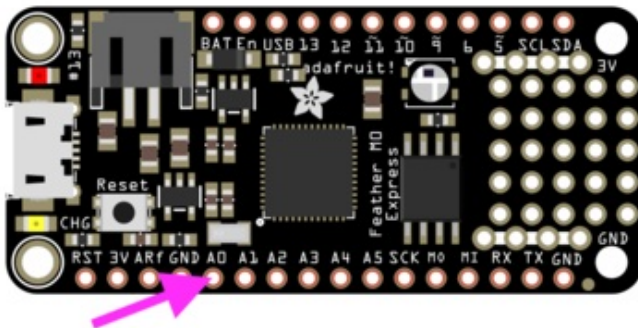
Remember, **A0** is labeled "1" on Trinket M0!

Gemma M0



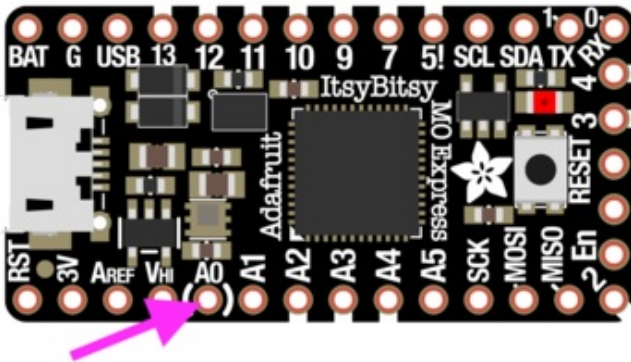
There are three pins on Gemma, in the form of alligator-clip-friendly pads, that work for touch input: **A0**, **A1** and **A2**.

Feather M0 Express



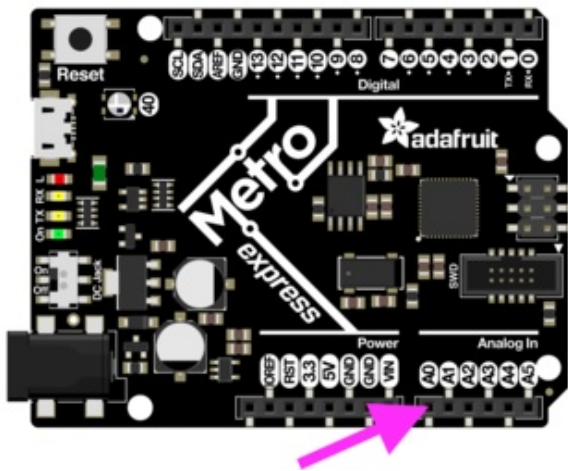
There are 6 pins on the Feather that have touch capability: **A0 - A5**.

ItsyBitsy M0 Express



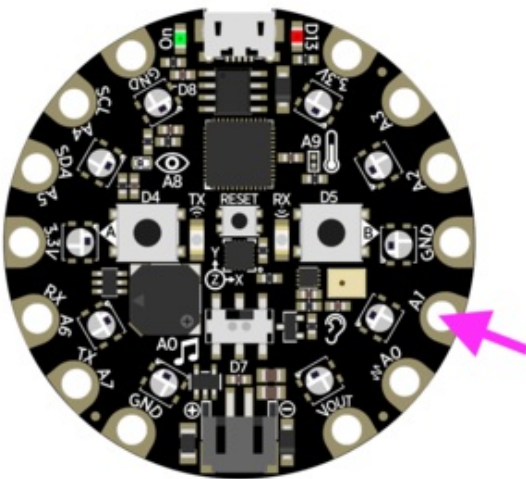
There are 6 pins on the ItsyBitsy that have touch capability: A0 - A5.

Metro M0 Express



There are 6 pins on the Metro that have touch capability: A0 - A5.

Circuit Playground Express

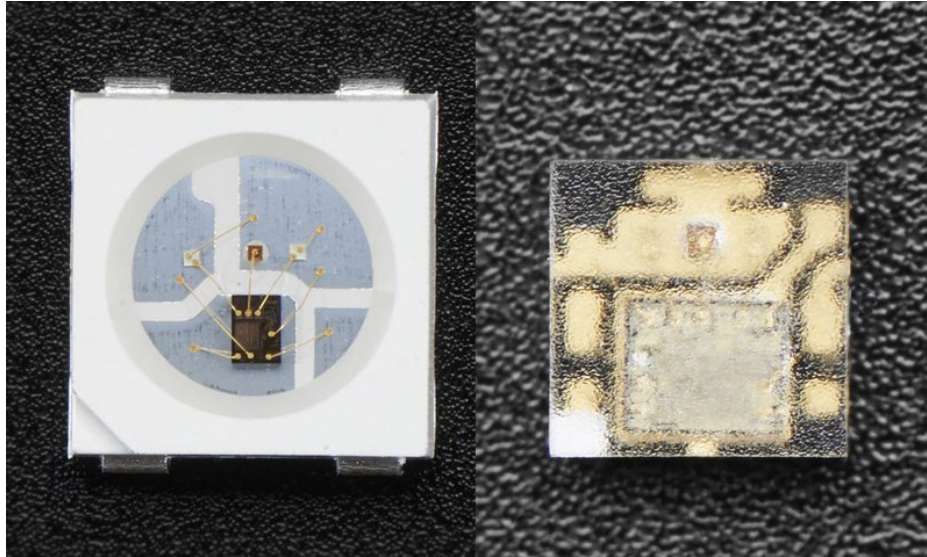


Circuit Playground Express has seven touch capable pins! You have **A1 - A7** available, in the form of alligator-clip-friendly pads. See the [CPX guide Cap Touch section \(https://adafru.it/ANC\)](https://adafru.it/ANC) for more information on using these pads for touch!

Remember: A0 does **NOT** have touch capabilities on CPX.

CircuitPython Internal RGB LED

Every board has a built in RGB LED. You can use CircuitPython to control the color and brightness of this LED. There are two different types of internal RGB LEDs: [DotStar](https://adafru.it/kDg) (<https://adafru.it/kDg>) and [NeoPixel](https://adafru.it/Bej) (<https://adafru.it/Bej>). This section covers both and explains which boards have which LED.



The first example will show you how to change the color and brightness of the internal RGB LED.

Copy and paste the code into `code.py` using your favorite editor, and save the file.

```
import time
import board

# For Trinket M0, Gemma M0, ItsyBitsy M0 Express, and ItsyBitsy M4 Express
import adafruit_dotstar
led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)
# For Feather M0 Express, Metro M0 Express, Metro M4 Express, and Circuit Playground Express
# import neopixel
# led = neopixel.NeoPixel(board.NEOPIXEL, 1)

led.brightness = 0.3

while True:
    led[0] = (255, 0, 0)
    time.sleep(0.5)
    led[0] = (0, 255, 0)
    time.sleep(0.5)
    led[0] = (0, 0, 255)
    time.sleep(0.5)
```

Create the LED

First, we create the LED object and attach it to the correct pin or pins. In the case of a NeoPixel, there is only one pin necessary, and we have called it `NEOPIXEL` for easier use. In the case of a DotStar, however, there are two pins necessary, and so we use the pin names `APA102_MOSI` and `APA102_SCK` to get it set up. Since we're using the

single onboard LED, the last thing we do is tell it that there's only `1` LED!

Trinket M0, Gemma M0, ItsyBitsy M0 Express, and ItsyBitsy M4 Express each have an onboard Dotstar LED, so no changes are needed to the initial version of the example.

Feather M0 Express, Feather M4 Express, Metro M0 Express, Metro M4 Express, and Circuit Playground Express each have an onboard NeoPixel LED, so you must comment out `import adafruit_dotstar` and `led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)`, and uncomment `import neopixel` and `led = neopixel.NeoPixel(board.NEOPIXEL, 1)`.



Remember: To "comment out" a line, put a `#` and a space before it. To "uncomment" a line, remove the `#` + space from the beginning of the line.

Brightness

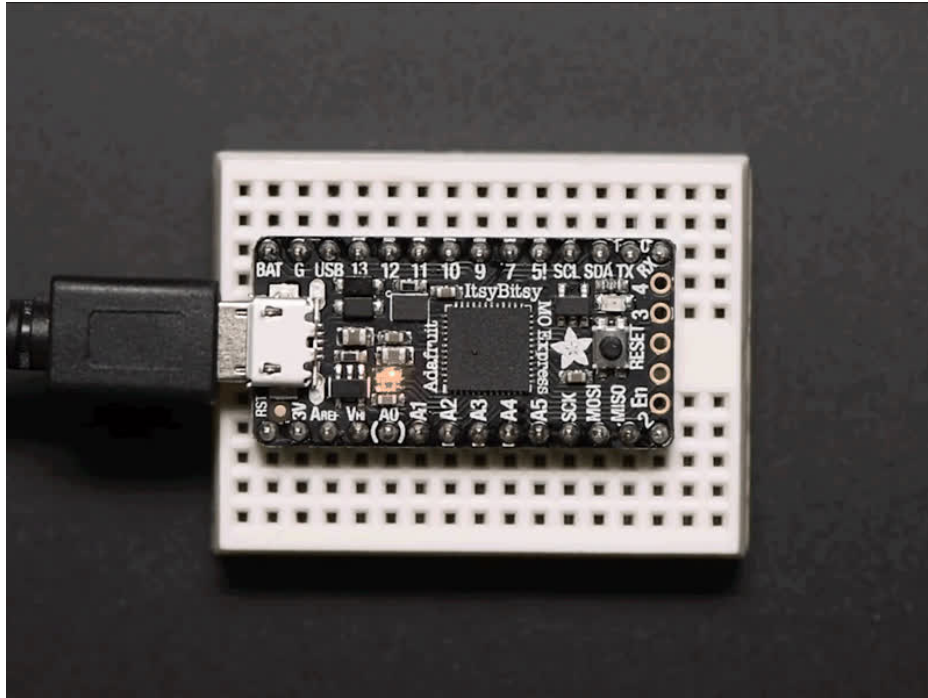
To set the brightness you simply use the `brightness` attribute. Brightness is set with a number between `0` and `1`, representative of a percent from 0% to 100%. So, `led.brightness = (0.3)` sets the LED brightness to 30%. The default brightness is `1` or 100%, and at its maximum, the LED is blindingly bright! You can set it lower if you choose.

Main Loop

LED colors are set using a combination of red, green, and blue, in the form of an **(R, G, B)** tuple. Each member of the tuple is set to a number between 0 and 255 that determines the amount of each color present. Red, green and blue in different combinations can create all the colors in the rainbow! So, for example, to set the LED to red, the tuple would be `(255, 0, 0)`, which has the maximum level of red, and no green or blue. Green would be `(0, 255, 0)`, etc. For the colors between, you set a combination, such as cyan which is `(0, 255, 255)`, with equal amounts of green and blue.

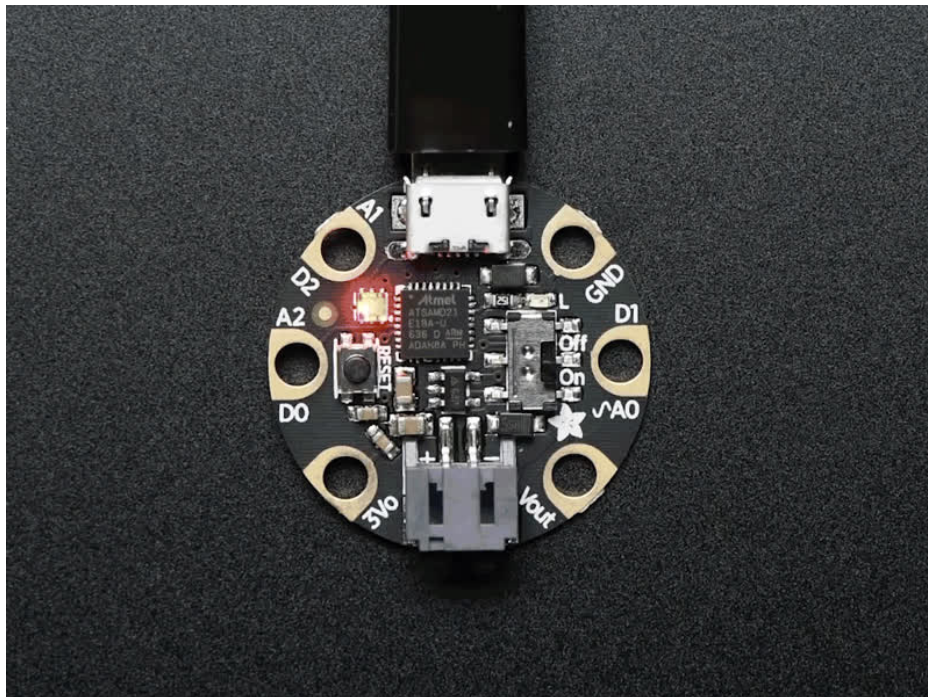
The main loop is quite simple. It sets the first LED to red using `(255, 0, 0)`, then green using `(0, 255, 0)`, and finally blue using `(0, 0, 255)`. Next, we give it a `time.sleep()` so it stays each color for a period of time. We chose `time.sleep(0.5)`, or half a second. Without the `time.sleep()` it'll flash really quickly and the colors will be difficult to see!

Note that we set `led[0]`. This means the first, and in the case of most of the boards, the only LED. In CircuitPython, counting starts at 0. So the first of any object, list, etc will be `0`!



Try changing the numbers in the tuples to change your LED to any color of the rainbow. Or, you can add more lines with different color tuples to add more colors to the sequence. Always add the `time.sleep()`, but try changing the amount of time to create different cycle animations!

Making Rainbows (Because Who Doesn't Love 'Em!)



Coding a rainbow effect involves a little math and a helper function called `wheel`. For details about how `wheel` works, see [this explanation here \(https://adafruit.it/Bek\)](https://adafruit.it/Bek)!

The last example shows how to do a rainbow animation on the internal RGB LED.

Copy and paste the code into `code.py` using your favorite editor, and save the file. **Remember to comment and uncomment the right lines for the board you're using**, as [explained above \(https://adafru.it/Bel\)](https://adafru.it/Bel).

```
import time
import board

# For Trinket M0, Gemma M0, ItsyBitsy M0 Express and ItsyBitsy M4 Express
import adafruit_dotstar
led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)
# For Feather M0 Express, Metro M0 Express, Metro M4 Express and Circuit Playground Express
# import neopixel
# led = neopixel.NeoPixel(board.NEOPIXEL, 1)

def wheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if pos < 0 or pos > 255:
        return 0, 0, 0
    if pos < 85:
        return int(255 - pos * 3), int(pos * 3), 0
    if pos < 170:
        pos -= 85
        return 0, int(255 - pos * 3), int(pos * 3)
    pos -= 170
    return int(pos * 3), 0, int(255 - (pos * 3))

led.brightness = 0.3

i = 0
while True:
    i = (i + 1) % 256 # run from 0 to 255
    led.fill(wheel(i))
    time.sleep(0.1)
```

We add the `wheel` function in after setup but before our main loop.

And right before our main loop, we assign the variable `i = 0`, so it's ready for use inside the loop.

The main loop contains some math that cycles `i` from `0` to `255` and around again repeatedly. We use this value to cycle `wheel()` through the rainbow!

The `time.sleep()` determines the speed at which the rainbow changes. Try a higher number for a slower rainbow or a lower number for a faster one!

Circuit Playground Express Rainbow

Note that here we use `led.fill` instead of `led[0]`. This means it turns on all the LEDs, which in the current code is only one. So why bother with `fill`? Well, you may have a Circuit Playground Express, which as you can see has TEN NeoPixel LEDs built in. The examples so far have only turned on the first one. If you'd like to do a rainbow on all ten LEDs, change the `1` in:


```
led = neopixel.NeoPixel(board.NEOPIXEL, 1)
```

to `10` so it reads:

```
led = neopixel.NeoPixel(board.NEOPIXEL, 10) .
```

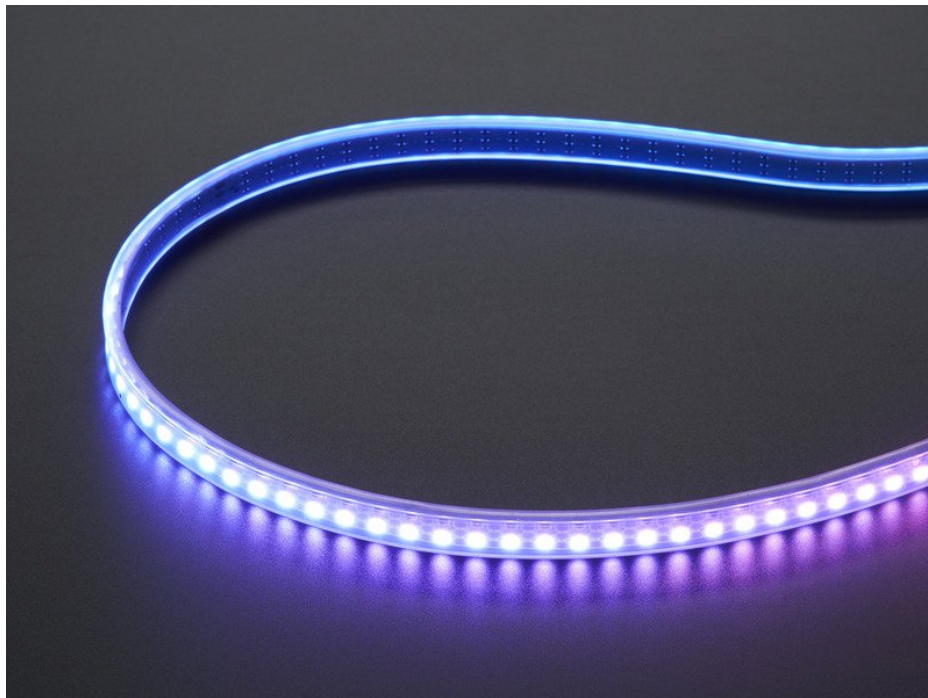
This tells the code to look for 10 LEDs instead of only 1. Now save the code and watch the rainbow go! You can make the same `1` to `10` change to the previous examples as well, and use `led.fill` to light up all the LEDs in the colors you chose! For more details, check out the [NeoPixel section of the CPX guide \(https://adafru.it/Bem\)](https://adafru.it/Bem)!

CircuitPython NeoPixel

NeoPixels are a revolutionary and ultra-popular way to add lights and color to your project. These stranded RGB lights have the controller inside the LED, so you just push the RGB data and the LEDs do all the work for you. They're a perfect match for CircuitPython!

You can drive 300 NeoPixel LEDs with brightness control (set `brightness=1.0` in object creation) and 1000 LEDs without. That's because to adjust the brightness we have to dynamically recreate the data-stream each write.

You'll need the `neopixel.mpy` library if you don't already have it in your `/lib` folder! You can get it from the [CircuitPython Library Bundle \(https://adafru.it/y8E\)](https://adafru.it/y8E). If you need help installing the library, check out the [CircuitPython Libraries page \(https://adafru.it/ABU\)](https://adafru.it/ABU).



Wiring It Up

You'll need to solder up your NeoPixels first. Verify your connection is on the **DATA INPUT** or **DIN** side. Plugging into the DATA OUT or DOUT side is a common mistake! The connections are labeled and some formats have arrows to indicate the direction the data must flow.

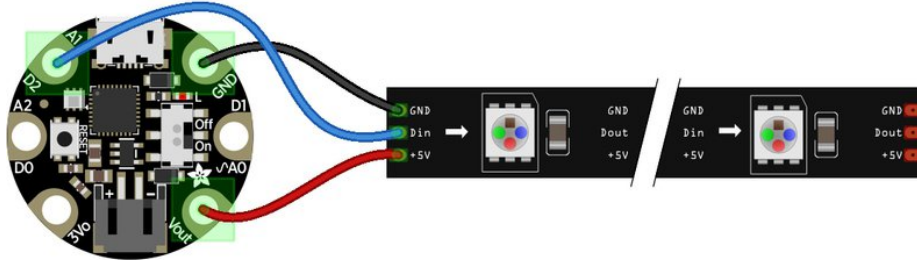
For powering the pixels from the board, the 3.3V regulator output can handle about 500mA peak which is about 50 pixels with 'average' use. If you want really bright lights and a lot of pixels, we recommend powering direct from an external power source.

- On Gemma M0 and Circuit Playground Express this is the **Vout** pad - that pad has direct power from USB or the battery, depending on which is higher voltage.
- On Trinket M0, Feather M0 Express, Feather M4 Express, ItsyBitsy M0 Express and ItsyBitsy M4 Express the **USB** or **BAT** pins will give you direct power from the USB port or battery.
- On Metro M0 Express and Metro M4 Express, use the **5V** pin regardless of whether it's powered via USB or the DC jack.

If the power to the NeoPixels is greater than 5.5V you may have some difficulty driving some strips, in which case you may need to lower the voltage to 4.5-5V or use a level shifter.



Do not use the VIN pin directly on Metro M0 Express or Metro M4 Express! The voltage can reach 9V and this can destroy your NeoPixels!



fritzing



Note that the wire ordering on your NeoPixel strip or shape may not exactly match the diagram above. Check the markings to verify which pin is DIN, 5V and GND

The Code

This example includes multiple visual effects. Copy and paste the code into `code.py` using your favorite editor, and save the file.

```
# CircuitPython demo - NeoPixel
import time
import board
import neopixel

pixel_pin = board.A1
num_pixels = 8

pixels = neopixel.NeoPixel(pixel_pin, num_pixels, brightness=0.3, auto_write=False)

def wheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if pos < 0 or pos > 255:
        return (0, 0, 0)
    if pos < 85:
        return (255 - pos * 3, pos * 3, 0)
    if pos < 170:
        pos -= 85
        return (0, 255 - pos * 3, pos * 3)
    pos -= 170
    return (pos * 3, 0, 255 - pos * 3)

def color_chase(color, wait):
    for i in range(num_pixels):
        pixels[i] = color
```

```

        time.sleep(wait)
        pixels.show()
    time.sleep(0.5)

def rainbow_cycle(wait):
    for j in range(255):
        for i in range(num_pixels):
            rc_index = (i * 256 // num_pixels) + j
            pixels[i] = wheel(rc_index & 255)
        pixels.show()
        time.sleep(wait)

RED = (255, 0, 0)
YELLOW = (255, 150, 0)
GREEN = (0, 255, 0)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
PURPLE = (180, 0, 255)

while True:
    pixels.fill(RED)
    pixels.show()
    # Increase or decrease to change the speed of the solid color change.
    time.sleep(1)
    pixels.fill(GREEN)
    pixels.show()
    time.sleep(1)
    pixels.fill(BLUE)
    pixels.show()
    time.sleep(1)

    color_chase(RED, 0.1) # Increase the number to slow down the color chase
    color_chase(YELLOW, 0.1)
    color_chase(GREEN, 0.1)
    color_chase(CYAN, 0.1)
    color_chase(BLUE, 0.1)
    color_chase(PURPLE, 0.1)

    rainbow_cycle(0) # Increase the number to slow down the rainbow

```

Create the LED

The first thing we'll do is create the LED object. The NeoPixel object has two required arguments and two optional arguments. You are required to set the pin you're using to drive your NeoPixels and provide the number of pixels you intend to use. You can optionally set `brightness` and `auto_write`.

NeoPixels can be driven by any pin. We've chosen **A1**. To set the pin, assign the variable `pixel_pin` to the pin you'd like to use, in our case `board.A1`.

To provide the number of pixels, assign the variable `num_pixels` to the number of pixels you'd like to use. In this example, we're using a strip of `8`.

We've chosen to set `brightness=0.3`, or 30%.

By default, `auto_write=True`, meaning any changes you make to your pixels will be sent automatically. Since `True` is the default, if you use that setting, you don't need to include it in your LED object at all. We've chosen to set `auto_write=False`. If you set `auto_write=False`, you must include `pixels.show()` each time you'd like to send data to your pixels. This makes your code more complicated, but it can make your LED animations faster!

NeoPixel Helpers

Next we've included a few helper functions to create the super fun visual effects found in this code. First is `wheel()` which we just learned with the [Internal RGB LED \(https://adafru.it/Bel\)](https://adafru.it/Bel). Then we have `color_chase()` which requires you to provide a `color` and the amount of time in seconds you'd like between each step of the chase. Next we have `rainbow_cycle()`, which requires you to provide the amount of time in seconds you'd like the animation to take. Last, we've included a list of variables for our colors. This makes it much easier if to reuse the colors anywhere in the code, as well as add more colors for use in multiple places. Assigning and using RGB colors is explained in [this section of the CircuitPython Internal RGB LED page \(https://adafru.it/Bel\)](https://adafru.it/Bel).

Main Loop

Thanks to our helpers, our main loop is quite simple. We include the code to set every NeoPixel we're using to red, green and blue for 1 second each. Then we call `color_chase()`, one time for each `color` on our list with `0.1` second delay between setting each subsequent LED the same color during the chase. Last we call `rainbow_cycle(0)`, which means the animation is as fast as it can be. Increase both of those numbers to slow down each animation!

Note that the longer your strip of LEDs, the longer it will take for the animations to complete.



We have a ton more information on general purpose NeoPixel know-how at our NeoPixel UberGuide <https://learn.adafruit.com/adafruit-neopixel-uberguide>

NeoPixel RGBW

NeoPixels are available in RGB, meaning there are three LEDs inside, red, green and blue. They're also available in RGBW, which includes four LEDs, red, green, blue and white. The code for RGBW NeoPixels is a little bit different than RGB.

If you run RGB code on RGBW NeoPixels, approximately 3/4 of the LEDs will light up and the LEDs will be the incorrect color even though they may appear to be changing. This is because NeoPixels require a piece of information for each available color (red, green, blue and possibly white).

Therefore, RGB LEDs require three pieces of information and RGBW LEDs require FOUR pieces of information to work. So when you create the LED object for RGBW LEDs, you'll include `bpp=4`, which sets bits-per-pixel to four (the four pieces of information!).

Then, you must include an extra number in every color tuple you create. For example, red will be `(255, 0, 0, 0)`. This is how you send the fourth piece of information. Check out the example below to see how our NeoPixel code looks for using with RGBW LEDs!

```
# CircuitPython demo - NeoPixel RGBW

import time
import board
import neopixel
```

```

pixel_pin = board.A1
num_pixels = 8

pixels = neopixel.NeoPixel(pixel_pin, num_pixels, brightness=0.3, auto_write=False,
                           pixel_order=(1, 0, 2, 3))

def wheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if pos < 0 or pos > 255:
        return (0, 0, 0, 0)
    if pos < 85:
        return (255 - pos * 3, pos * 3, 0, 0)
    if pos < 170:
        pos -= 85
        return (0, 255 - pos * 3, pos * 3, 0)
    pos -= 170
    return (pos * 3, 0, 255 - pos * 3, 0)

def color_chase(color, wait):
    for i in range(num_pixels):
        pixels[i] = color
        time.sleep(wait)
        pixels.show()
    time.sleep(0.5)

def rainbow_cycle(wait):
    for j in range(255):
        for i in range(num_pixels):
            rc_index = (i * 256 // num_pixels) + j
            pixels[i] = wheel(rc_index & 255)
        pixels.show()
        time.sleep(wait)

RED = (255, 0, 0, 0)
YELLOW = (255, 150, 0, 0)
GREEN = (0, 255, 0, 0)
CYAN = (0, 255, 255, 0)
BLUE = (0, 0, 255, 0)
PURPLE = (180, 0, 255, 0)

while True:
    pixels.fill(RED)
    pixels.show()
    # Increase or decrease to change the speed of the solid color change.
    time.sleep(1)
    pixels.fill(GREEN)
    pixels.show()
    time.sleep(1)
    pixels.fill(BLUE)
    pixels.show()
    time.sleep(1)

    color_chase(RED, 0.1) # Increase the number to slow down the color chase
    color_chase(YELLOW, 0.1)
    color_chase(GREEN, 0.1)

```

```
color_chase(GREEN, 0.1),
color_chase(CYAN, 0.1)
color_chase(BLUE, 0.1)
color_chase(PURPLE, 0.1)

rainbow_cycle(0) # Increase the number to slow down the rainbow
```

Read the Docs

For a more in depth look at what `neopixel` can do, check out [NeoPixel on Read the Docs \(https://adafru.it/C5m\)](https://adafru.it/C5m).

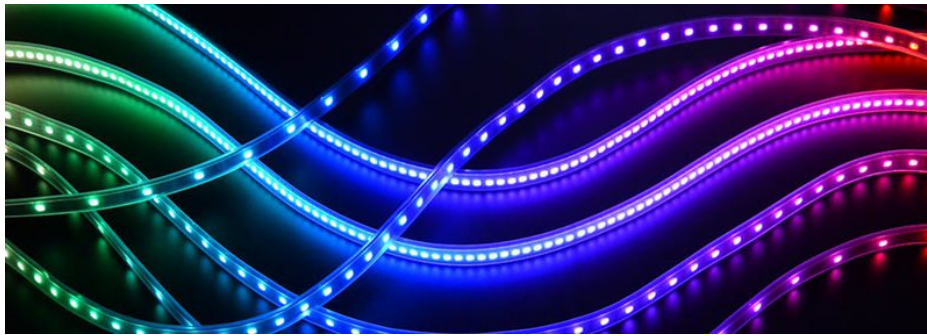
CircuitPython DotStar

DotStars use two wires, unlike NeoPixel's one wire. They're very similar but you can write to DotStars much faster with hardware SPI *and* they have a faster PWM cycle so they are better for light painting.

Any pins can be used **but** if the two pins can form a hardware SPI port, the library will automatically switch over to hardware SPI. If you use hardware SPI then you'll get 4 MHz clock rate (that would mean updating a 64 pixel strand in about 500uS - that's 0.0005 seconds). If you use non-hardware SPI pins you'll drop down to about 3KHz, 1000 times as slow!

You can drive 300 DotStar LEDs with brightness control (set `brightness=1.0` in object creation) and 1000 LEDs without. That's because to adjust the brightness we have to dynamically recreate the data-stream each write.

You'll need the `adafruit_dotstar.mpy` library if you don't already have it in your `/lib` folder! You can get it from the [CircuitPython Library Bundle \(https://adafru.it/y8E\)](https://adafru.it/y8E). If you need help installing the library, check out the [CircuitPython Libraries page \(https://adafru.it/ABU\)](https://adafru.it/ABU).



Wire It Up

You'll need to solder up your DotStars first. Verify your connection is on the **DATA INPUT** or **DI** and **CLOCK INPUT** or **CI** side. Plugging into the DATA OUT/DO or CLOCK OUT/CO side is a common mistake! The connections are labeled and some formats have arrows to indicate the direction the data must flow. Always verify your wiring with a visual inspection, as the order of the connections can differ from strip to strip!

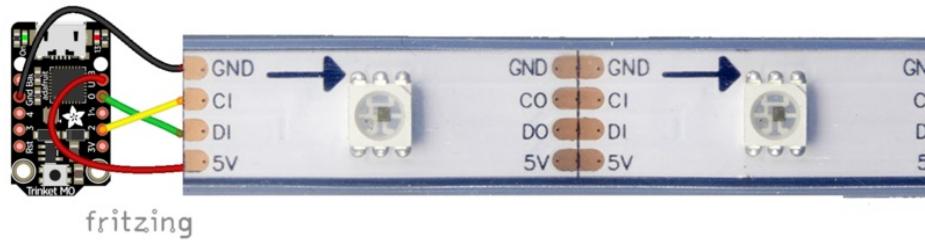
For powering the pixels from the board, the 3.3V regulator output can handle about 500mA peak which is about 50 pixels with 'average' use. If you want really bright lights and a lot of pixels, we recommend powering direct from an external power source.

- On Gemma M0 and Circuit Playground Express this is the **Vout** pad - that pad has direct power from USB or the battery, depending on which is higher voltage.
- On Trinket M0, Feather M0 Express, Feather M4 Express, ItsyBitsy M0 Express and ItsyBitsy M4 Express the **USB** or **BAT** pins will give you direct power from the USB port or battery.
- On Metro M0 Express and Metro M4 Express, use the **5V** pin regardless of whether it's powered via USB or the DC jack.

If the power to the DotStars is greater than 5.5V you may have some difficulty driving some strips, in which case you may need to lower the voltage to 4.5-5V or use a level shifter.



Do not use the VIN pin directly on Metro M0 Express or Metro M4 Express! The voltage can reach 9V and this can destroy your DotStars!



Note that the wire ordering on your DotStar strip or shape may not exactly match the diagram above. Check the markings to verify which pin is DIN, CIN, 5V and GND

The Code

This example includes multiple visual effects. Copy and paste the code into `code.py` using your favorite editor, and save the file.

```
# CircuitPython demo - Dotstar
import time
import adafruit_dotstar
import board

num_pixels = 30
pixels = adafruit_dotstar.DotStar(board.A1, board.A2, num_pixels, brightness=0.1, auto_write=False)

def wheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if pos < 0 or pos > 255:
        return (0, 0, 0)
    if pos < 85:
        return (255 - pos * 3, pos * 3, 0)
    if pos < 170:
        pos -= 85
        return (0, 255 - pos * 3, pos * 3)
    pos -= 170
    return (pos * 3, 0, 255 - pos * 3)

def color_fill(color, wait):
    pixels.fill(color)
    pixels.show()
    time.sleep(wait)

def slice_alternating(wait):
    pixels[::2] = [RED] * (num_pixels // 2)
    pixels.show()
    time.sleep(wait)
    pixels[1::2] = [ORANGE] * (num_pixels // 2)
    pixels.show()
```

```

pixels.show()
time.sleep(wait)
pixels[::2] = [YELLOW] * (num_pixels // 2)
pixels.show()
time.sleep(wait)
pixels[1::2] = [GREEN] * (num_pixels // 2)
pixels.show()
time.sleep(wait)
pixels[::2] = [TEAL] * (num_pixels // 2)
pixels.show()
time.sleep(wait)
pixels[1::2] = [CYAN] * (num_pixels // 2)
pixels.show()
time.sleep(wait)
pixels[::2] = [BLUE] * (num_pixels // 2)
pixels.show()
time.sleep(wait)
pixels[1::2] = [PURPLE] * (num_pixels // 2)
pixels.show()
time.sleep(wait)
pixels[::2] = [MAGENTA] * (num_pixels // 2)
pixels.show()
time.sleep(wait)
pixels[1::2] = [WHITE] * (num_pixels // 2)
pixels.show()
time.sleep(wait)

```

```

def slice_rainbow(wait):
    pixels[::6] = [RED] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[1::6] = [ORANGE] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[2::6] = [YELLOW] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[3::6] = [GREEN] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[4::6] = [BLUE] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[5::6] = [PURPLE] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)

```

```

def rainbow_cycle(wait):
    for j in range(255):
        for i in range(num_pixels):
            rc_index = (i * 256 // num_pixels) + j
            pixels[i] = wheel(rc_index & 255)
        pixels.show()
        time.sleep(wait)

```

```

RED = (255, 0, 0)
YELLOW = (255, 150, 0)

```

```

ORANGE = (255, 40, 0)
GREEN = (0, 255, 0)
TEAL = (0, 255, 120)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
PURPLE = (180, 0, 255)
MAGENTA = (255, 0, 20)
WHITE = (255, 255, 255)

while True:
    # Change this number to change how long it stays on each solid color.
    color_fill(RED, 0.5)
    color_fill(YELLOW, 0.5)
    color_fill(ORANGE, 0.5)
    color_fill(GREEN, 0.5)
    color_fill(TEAL, 0.5)
    color_fill(CYAN, 0.5)
    color_fill(BLUE, 0.5)
    color_fill(PURPLE, 0.5)
    color_fill(MAGENTA, 0.5)
    color_fill(WHITE, 0.5)

    # Increase or decrease this to speed up or slow down the animation.
    slice_alternating(0.1)

    color_fill(WHITE, 0.5)

    # Increase or decrease this to speed up or slow down the animation.
    slice_rainbow(0.1)

    time.sleep(0.5)

    # Increase this number to slow down the rainbow animation.
    rainbow_cycle(0)

```



We've chosen pins A1 and A2, but these are not SPI pins on all boards. DotStars respond faster when using hardware SPI!

Create the LED

The first thing we'll do is create the LED object. The DotStar object has three required arguments and two optional arguments. You are required to set the pin you're using for data, set the pin you'll be using for clock, and provide the number of pixels you intend to use. You can optionally set `brightness` and `auto_write`.

DotStars can be driven by any two pins. We've chosen **A1** for clock and **A2** for data. To set the pins, include the pin names at the beginning of the object creation, in this case `board.A1` and `board.A2`.

To provide the number of pixels, assign the variable `num_pixels` to the number of pixels you'd like to use. In this example, we're using a strip of `72`.

We've chosen to set `brightness=0.1`, or 10%.

By default, `auto_write=True`, meaning any changes you make to your pixels will be sent automatically. Since `True` is the default, if you use that setting, you don't need to include it in your LED object at all. We've chosen to

set `auto_write=False`. If you set `auto_write=False`, you must include `pixels.show()` each time you'd like to send data to your pixels. This makes your code more complicated, but it can make your LED animations faster!

DotStar Helpers

We've included a few helper functions to create the super fun visual effects found in this code.

First is `wheel()` which we just learned with the [Internal RGB LED \(https://adafru.it/Bel\)](https://adafru.it/Bel). Then we have `color_fill()` which requires you to provide a `color` and the length of time you'd like it to be displayed. Next, are `slice_alternating()`, `slice_rainbow()`, and `rainbow_cycle()` which require you to provide the amount of time in seconds you'd between each step of the animation.

Last, we've included a list of variables for our colors. This makes it much easier if to reuse the colors anywhere in the code, as well as add more colors for use in multiple places. Assigning and using RGB colors is explained in [this section of the CircuitPython Internal RGB LED page \(https://adafru.it/Bel\)](https://adafru.it/Bel).

The two slice helpers utilise a nifty feature of the DotStar library that allows us to use math to light up LEDs in repeating patterns. `slice_alternating()` first lights up the even number LEDs and then the odd number LEDs and repeats this back and forth. `slice_rainbow()` lights up every sixth LED with one of the six rainbow colors until the strip is filled. Both use our handy color variables. This slice code only works when the total number of LEDs is divisible by the slice size, in our case 2 and 6. DotStars come in strips of 30, 60, 72, and 144, all of which are divisible by 2 and 6. In the event that you cut them into different sized strips, the code in this example may not work without modification. However, as long as you provide a total number of LEDs that is divisible by the slices, the code will work.

Main Loop

Our main loop begins by calling `color_fill()` once for each `color` on our list and sets each to hold for `0.5` seconds. You can change this number to change how fast each color is displayed. Next, we call `slice_alternating(0.1)`, which means there's a 0.1 second delay between each change in the animation. Then, we fill the strip white to create a clean backdrop for the rainbow to display. Then, we call `slice_rainbow(0.1)`, for a 0.1 second delay in the animation. Last we call `rainbow_cycle(0)`, which means it's as fast as it can possibly be. Increase or decrease either of these numbers to speed up or slow down the animations!

Note that the longer your strip of LEDs is, the longer it will take for the animations to complete.



We have a ton more information on general purpose DotStar know-how at our DotStar UberGuide <https://learn.adafruit.com/adafruit-dotstar-leds>

Is it SPI?

We explained at the beginning of this section that the LEDs respond faster if you're using hardware SPI. On some of the boards, there are HW SPI pins directly available in the form of MOSI and SCK. However, hardware SPI is available on more than just those pins. But, how can you figure out which? Easy! We wrote a handy script.

We chose pins **A1** and **A2** for our example code. To see if these are hardware SPI on the board you're using, copy and paste the code into `code.py` using your favorite editor, and save the file. Then connect to the serial console to see the results.

To check if other pin combinations have hardware SPI, change the pin names on the line reading: `if is_hardware_SPI(board.A1, board.A2):` to the pins you want to use. Then, check the results in the serial console. Super simple!

```
import board
import busio

def is_hardware_spi(clock_pin, data_pin):
    try:
        p = busio.SPI(clock_pin, data_pin)
        p.deinit()
        return True
    except ValueError:
        return False

# Provide the two pins you intend to use.
if is_hardware_spi(board.A1, board.A2):
    print("This pin combination is hardware SPI!")
else:
    print("This pin combination isn't hardware SPI.")
```

Read the Docs

For a more in depth look at what [dotstar](#) can do, check out [DotStar on Read the Docs \(https://adafru.it/C4d\)](https://adafru.it/C4d).

CircuitPython UART Serial

In addition to the USB-serial connection you use for the REPL, there is also a *hardware* UART you can use. This is handy to talk to UART devices like GPSs, some sensors, or other microcontrollers!

This quick-start example shows how you can create a UART device for communicating with hardware serial devices.

To use this example, you'll need something to generate the UART data. We've used a GPS! Note that the GPS will give you UART data without getting a fix on your location. You can use this example right from your desk! You'll have data to read, it simply won't include your actual location.

You'll need the `adafruit_bus_device` library folder if you don't already have it in your `/lib` folder! You can get it from the [CircuitPython Library Bundle \(https://adafru.it/y8E\)](https://adafru.it/y8E). If you need help installing the library, check out the [CircuitPython Libraries page \(https://adafru.it/ABU\)](https://adafru.it/ABU).

Copy and paste the code into `code.py` using your favorite editor, and save the file.

```
# CircuitPython Demo - USB/Serial echo

import board
import busio
import digitalio

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

uart = busio.UART(board.TX, board.RX, baudrate=9600)

while True:
    data = uart.read(32) # read up to 32 bytes
    # print(data) # this is a bytearray type

    if data is not None:
        led.value = True

        # convert bytearray to string
        data_string = ''.join([chr(b) for b in data])
        print(data_string, end="")

        led.value = False
```

The Code

First we create the UART object. We provide the pins we'd like to use, `board.TX` and `board.RX`, and we set the `baudrate=9600`. While these pins are labeled on most of the boards, be aware that RX and TX are not labeled on Gemma, and are labeled on the bottom of Trinket. See the diagrams below for help with finding the correct pins on your board.

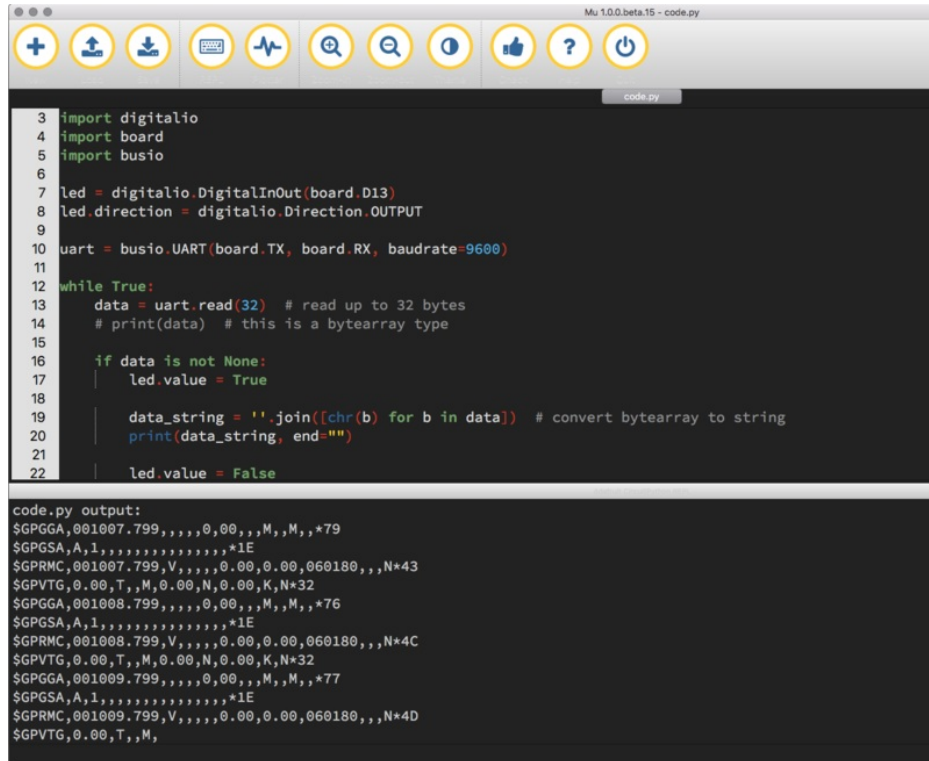
Once the object is created you read data in with `read(numbytes)` where you can specify the max number of bytes. It will return a byte array type object if anything was received already. Note it will always return immediately because there is an internal buffer! So read as much data as you can 'digest'.

If there is no data available, `read()` will return `None`, so check for that before continuing.

The data that is returned is in a byte array, if you want to convert it to a string, you can use this handy line of code which will run `chr()` on each byte:

```
datastr = ''.join([chr(b) for b in data]) # convert bytearray to string
```

Your results will look something like this:



```
3 import digitalio
4 import board
5 import busio
6
7 led = digitalio.DigitalInOut(board.D13)
8 led.direction = digitalio.Direction.OUTPUT
9
10 uart = busio.UART(board.TX, board.RX, baudrate=9600)
11
12 while True:
13     data = uart.read(32) # read up to 32 bytes
14     # print(data) # this is a bytearray type
15
16     if data is not None:
17         led_value = True
18
19         data_string = ''.join([chr(b) for b in data]) # convert bytearray to string
20         print(data_string, end="")
21
22     led_value = False
```

code.py output:

```
$GPGGA,001007.799,,,0,00,,M,,M,,*79
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPRMC,001007.799,V,,,0.00,0.00,060180,,,N*43
$GPVTG,0.00,T,,M,0.00,N,0.00,K,N*32
$GPGGA,001008.799,,,0,00,,M,,M,,*76
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPRMC,001008.799,V,,,0.00,0.00,060180,,,N*4C
$GPVTG,0.00,T,,M,0.00,N,0.00,K,N*32
$GPGGA,001009.799,,,0,00,,M,,M,,*77
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPRMC,001009.799,V,,,0.00,0.00,060180,,,N*4D
$GPVTG,0.00,T,,M,
```

For more information about the data you're reading and the Ultimate GPS, check out the Ultimate GPS guide: <https://learn.adafruit.com/adafruit-ultimate-gps>

Wire It Up

You'll need a couple of things to connect the GPS to your board.

For Gemma M0 and Circuit Playground Express, you can use alligator clips to connect to the Flora Ultimate GPS Module.

For Trinket M0, Feather M0 Express, Metro M0 Express and ItsyBitsy M0 Express, you'll need a breadboard and jumper wires to connect to the Ultimate GPS Breakout.

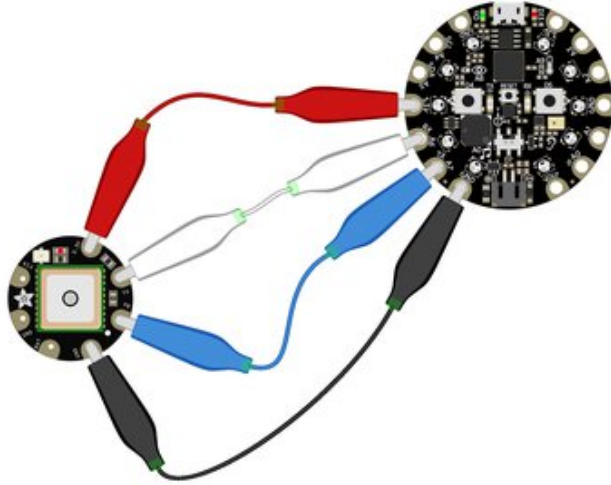
We've included diagrams show you how to connect the GPS to your board. In these diagrams, the wire colors match the same pins on each board.

- The **black** wire connects between the **ground** pins.
- The **red** wire connects between the **power** pins on the GPS and your board.
- The **blue** wire connects from **TX** on the GPS to **RX** on your board.
- The **white** wire connects from **RX** on the GPS to **TX** on your board.

Check out the list below for a diagram of your specific board!

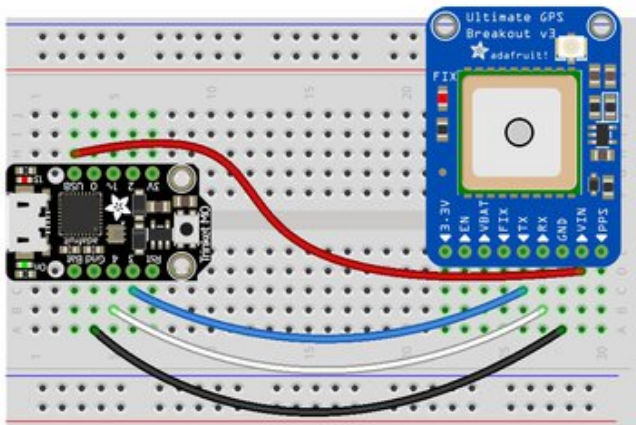


Watch out! A common mixup with UART serial is that RX on one board connects to TX on the other! However, sometimes boards have RX labeled TX and vice versa. So, you'll want to start with RX connected to TX, but if that doesn't work, try the other way around!



Circuit Playground Express

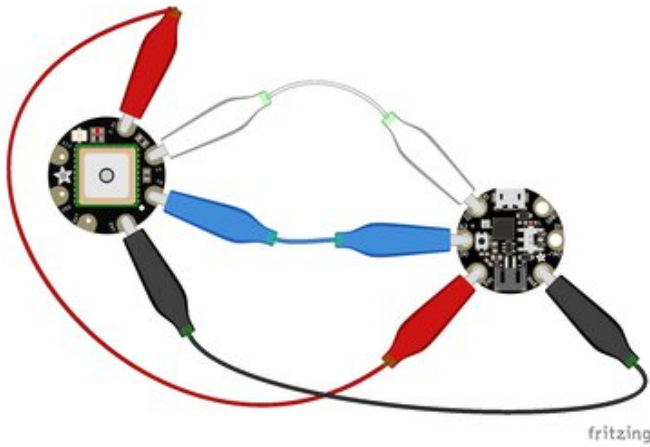
- Connect **3.3v** on your CPX to **3.3v** on your GPS.
- Connect **GND** on your CPX to **GND** on your GPS.
- Connect **RX/A6** on your CPX to **TX** on your GPS.
- Connect **TX/A7** on your CPX to **RX** on your GPS.



Trinket M0

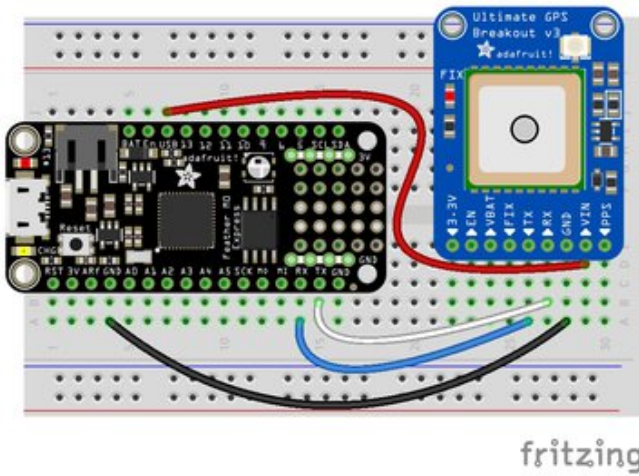
- Connect **USB** on the Trinket to **VIN** on the GPS.
- Connect **Gnd** on the Trinket to **GND** on the GPS.
- Connect **D3** on the Trinket to **TX** on the GPS.
- Connect **D4** on the Trinket to **RX** on the GPS.

fritzing



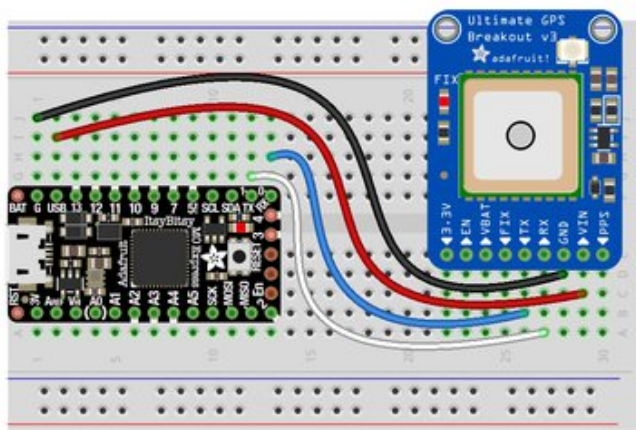
Gemma M0

- Connect **3v0** on the Gemma to **3.3v** on the GPS.
- Connect **GND** on the Gemma to **GND** on the GPS.
- Connect **A1/D2** on the Gemma to **TX** on the GPS.
- Connect **A2/D0** on the Gemma to **RX** on the GPS.



Feather M0 Express and Feather M4 Express

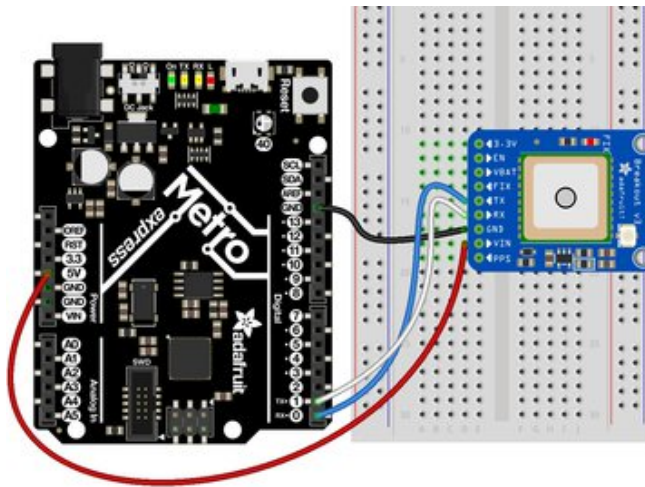
- Connect **USB** on the Feather to **VIN** on the GPS.
- Connect **GND** on the Feather to **GND** on the GPS.
- Connect **RX** on the Feather to **TX** on the GPS.
- Connect **TX** on the Feather to **RX** on the GPS.



fritzing

ItsyBitsy M0 Express and ItsyBitsy M4 Express

- Connect **USB** on the ItsyBitsy to **VIN** on the GPS.
- Connect **G** on the ItsyBitsy to **GND** on the GPS.
- Connect **RX/0** on the ItsyBitsy to **TX** on the GPS.
- Connect **TX/1** on the ItsyBitsy to **RX** on the GPS.



Metro M0 Express and Metro M4 Express

- Connect **5V** on the Metro to **VIN** on the GPS.
- Connect **GND** on the Metro to **GND** on the GPS.
- Connect **RX/D0** on the Metro to **TX** on the GPS.
- Connect **TX/D1** on the Metro to **RX** on the GPS.

Where's my UART?

On the SAMD21, we have the flexibility of using a wide range of pins for UART. Compare this to some chips like the ESP8266 with *fixed* UART pins. The good news is you can use many but not *all* pins. Given the large number of SAMD boards we have, its impossible to guarantee anything other than the labeled 'TX' and 'RX'. So, if you want some other setup, or multiple UARTs, how will you find those pins? Easy! We've written a handy script.

All you need to do is copy this file to your board, rename it **code.py**, connect to the serial console and check out the output! The results print out a nice handy list of RX and TX pin pairs that you can use.

These are the results from a Trinket M0, your output may vary and it might be *very* long. [For more details about UARTs and SERCOMs check out our detailed guide here \(https://adafru.it/Ben\)](https://adafru.it/Ben)

```

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
RX pin: board.D2      TX pin: board.D0
RX pin: board.D4      TX pin: board.D0
RX pin: board.D3      TX pin: board.D0
RX pin: board.D13     TX pin: board.D0
RX pin: board.D0      TX pin: board.D4
RX pin: board.D2      TX pin: board.D4
RX pin: board.D3      TX pin: board.D4
RX pin: board.D0      TX pin: board.D13
RX pin: board.D2      TX pin: board.D13
RX pin: board.D3      TX pin: board.D13

```

```

import board
import busio
from microcontroller import Pin

def is_hardware_uart(tx, rx):
    try:
        p = busio.UART(tx, rx)
        p.deinit()
        return True
    except ValueError:
        return False

def get_unique_pins():
    exclude = ['NEOPIXEL', 'APA102_MOSI', 'APA102_SCK']
    pins = [pin for pin in [
        getattr(board, p) for p in dir(board) if p not in exclude]
        if isinstance(pin, Pin)]
    unique = []
    for p in pins:
        if p not in unique:
            unique.append(p)
    return unique

for tx_pin in get_unique_pins():
    for rx_pin in get_unique_pins():
        if rx_pin is tx_pin:
            continue
        else:
            if is_hardware_uart(tx_pin, rx_pin):
                print("RX pin:", rx_pin, "\t TX pin:", tx_pin)
            else:
                pass

```

Trinket M0: Create UART before I2C

On the Trinket M0 (only), if you are using both `busio.UART` and `busio.I2C`, you must create the UART object first, e.g.:

```

>>> import board,busio
>>> uart = busio.UART(board.TX, board.RX)
>>> i2c = busio.I2C(board.SCL, board.SDA)

```

Creating `busio.I2C` first does not work:

```
>>> import board,busio
>>> i2c = busio.I2C(board.SCL, board.SDA)
>>> uart = busio.UART(board.TX, board.RX)
Traceback (most recent call last):
File "", line 1, in
ValueError: Invalid pins
```

CircuitPython I2C

I2C is a 2-wire protocol for communicating with simple sensors and devices, meaning it uses two connections for transmitting and receiving data. There are many I2C devices available and they're really easy to use with CircuitPython. We have libraries available for many I2C devices in the [library bundle \(https://adafru.it/uap\)](https://adafru.it/uap). (If you don't see the sensor you're looking for, keep checking back, more are being written all the time!)

In this section, we're going to do is learn how to scan the I2C bus for all connected devices. Then we're going to learn how to interact with an I2C device.

We'll be using the TSL2561, a common, low-cost light sensor. While the exact code we're running is specific to the TSL2561 the overall process is the same for just about any I2C sensor or device.

You'll need the `adafruit_tsl2561.mpy` library and `adafruit_bus_device` library folder if you don't already have it in your `/lib` folder! You can get it from the [CircuitPython Library Bundle \(https://adafru.it/y8E\)](https://adafru.it/y8E). If you need help installing the library, check out the [CircuitPython Libraries page \(https://adafru.it/ABU\)](https://adafru.it/ABU).

These examples will use the TSL2561 lux sensor Flora and breakout. The first thing you'll want to do is get the sensor connected so your board has I2C to talk to.

Wire It Up

You'll need a couple of things to connect the TSL2561 to your board.

For Gemma M0 and Circuit Playground Express, you can use use alligator clips to connect to the Flora TSL2561 Lux Sensor.

For Trinket M0, Feather M0 Express, Metro M0 Express and ItsyBitsy M0 Express, you'll need a breadboard and jumper wires to connect to the TSL2561 Lux Sensor breakout board.

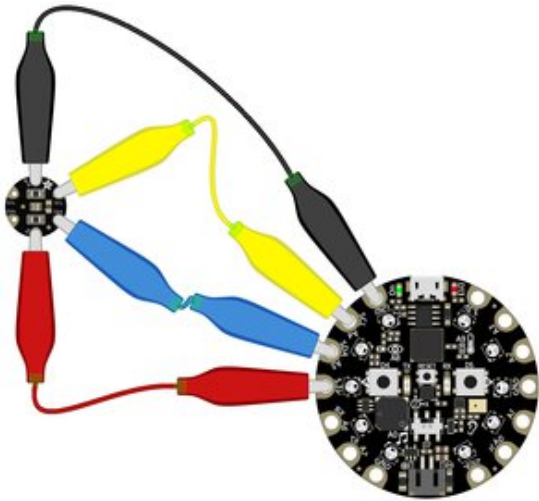
We've included diagrams show you how to connect the TSL2561 to your board. In these diagrams, the wire colors match the same pins on each board.

- The **black** wire connects between the **ground** pins.
- The **red** wire connects between the **power** pins on the TSL2561 and your board.
- The **yellow** wire connects from **SCL** on the TSL2561 to **SCL** on your board.
- The **blue** wire connects from **SDA** on the TSL2561 to **SDA** on your board.

Check out the list below for a diagram of your specific board!

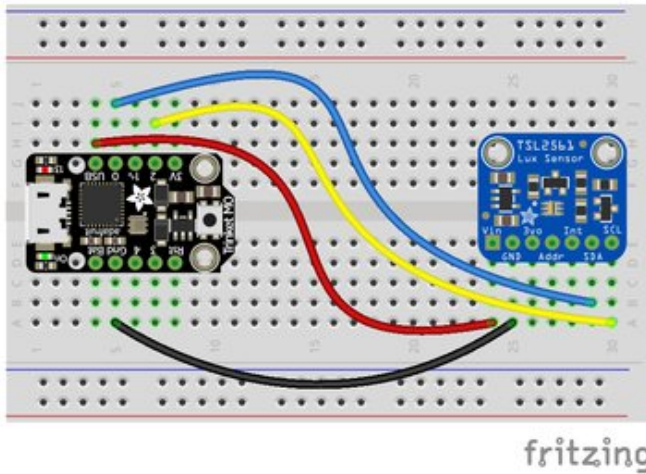


Be aware that the Adafruit microcontroller boards do not have I2C pullup resistors built in! All of the Adafruit breakouts do, but if you're building your own board or using a non-Adafruit breakout, you must add 2.2K-10K ohm pullups on both SDA and SCL to the 3.3V.



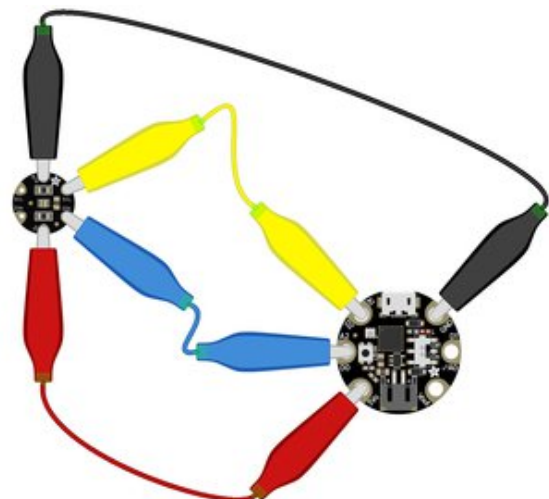
Circuit Playground Express

- Connect **3.3v** on your CPX to **3.3v** on your TSL2561.
- Connect **GND** on your CPX to **GND** on your TSL2561.
- Connect **SCL/A4** on your CPX to **SCL** on your TSL2561.
- Connect **SDL/A5** on your CPX to **SDA** on your TSL2561.



Trinket M0

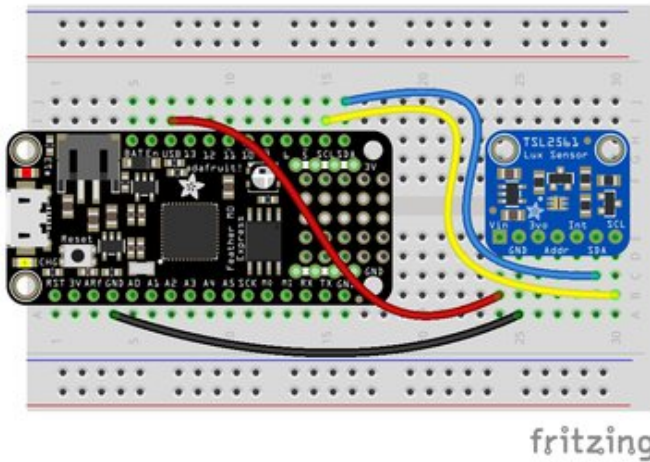
- Connect **USB** on the Trinket to **VIN** on the TSL2561.
- Connect **Gnd** on the Trinket to **GND** on the TSL2561.
- Connect **D2** on the Trinket to **SCL** on the TSL2561.
- Connect **D0** on the Trinket to **SDA** on the TSL2561.



Gemma M0

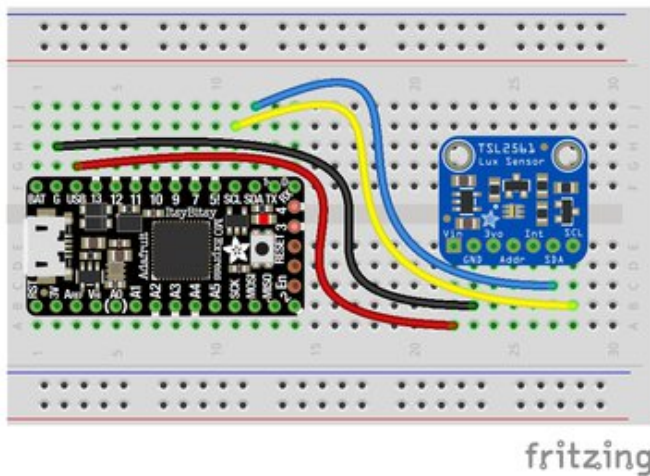
- Connect **3vo** on the Gemma to **3V** on the TSL2561.
- Connect **GND** on the Gemma to **GND** on the TSL2561.
- Connect **A1/D2** on the Gemma to **SCL** on the TSL2561.
- Connect **A2/D0** on the Gemma to **SDA** on the TSL2561.

Feather M0 Express and Feather M4 Express



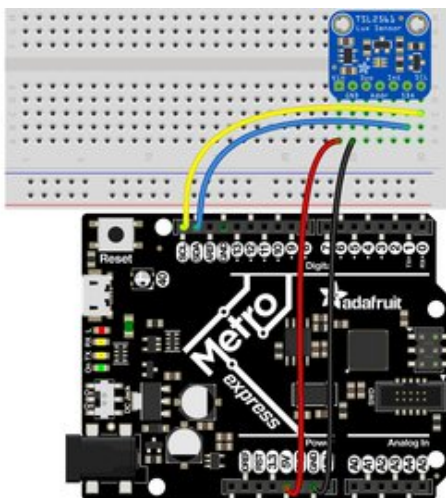
- Connect **USB** on the Feather to **VIN** on the TSL2561.
- Connect **GND** on the Feather to **GND** on the TSL2561.
- Connect **SCL** on the Feather to **SCL** on the TSL2561.
- Connect **SDA** on the Feather to **SDA** on the TSL2561.

ItsyBitsy M0 Express and ItsyBitsy M4 Express



- Connect **USB** on the ItsyBitsy to **VIN** on the TSL2561
- Connect **G** on the ItsyBitsy to **GND** on the TSL2561.
- Connect **SCL** on the ItsyBitsy to **SCL** on the TSL2561.
- Connect **SDA** on the ItsyBitsy to **SDA** on the TSL2561.

Metro M0 Express and Metro M4 Express



- Connect **5V** on the Metro to **VIN** on the TSL2561.
- Connect **GND** on the Metro to **GND** on the TSL2561.
- Connect **SCL** on the Metro to **SCL** on the TSL2561.
- Connect **SDA** on the Metro to **SDA** on the TSL2561.

Find Your Sensor

The first thing you'll want to do after getting the sensor wired up, is make sure it's wired correctly. We're going to do an I2C scan to see if the board is detected, and if it is, print out its I2C address.

Copy and paste the code into `code.py` using your favorite editor, and save the file.

```
# CircuitPython demo - I2C scan

import time

import board
import busio

i2c = busio.I2C(board.SCL, board.SDA)

while not i2c.try_lock():
    pass

while True:
    print("I2C addresses found:", [hex(device_address)
                                   for device_address in i2c.scan()])
    time.sleep(2)
```

First we create the `i2c` object and provide the I2C pins, `board.SCL` and `board.SDA`.

To be able to scan it, we need to lock the I2C down so the only thing accessing it is the code. So next we include a loop that waits until I2C is locked and then continues on to the scan function.

Last, we have the loop that runs the actual scan, `i2c.scan()`. Because I2C typically refers to addresses in hex form, we've included this bit of code that formats the results into hex format: `[hex(device_address) for device_address in i2c.scan()]`.

Open the serial console to see the results! The code prints out an array of addresses. We've connected the TSL2561 which has a 7-bit I2C address of 0x39. The result for this sensor is `I2C addresses found: ['0x39']`. If no addresses are returned, refer back to the wiring diagrams to make sure you've wired up your sensor correctly.

I2C Sensor Data

Now we know for certain that our sensor is connected and ready to go. Let's find out how to get the data from our sensor!

Copy and paste the code into `code.py` using your favorite editor, and save the file.


```

# CircuitPython Demo - I2C sensor

import time

import adafruit_tsl2561
import board
import busio

i2c = busio.I2C(board.SCL, board.SDA)

# Lock the I2C device before we try to scan
while not i2c.try_lock():
    pass
# Print the addresses found once
print("I2C addresses found:", [hex(device_address)
                               for device_address in i2c.scan()])

# Unlock I2C now that we're done scanning.
i2c.unlock()

# Create library object on our I2C port
tsl2561 = adafruit_tsl2561.TSL2561(i2c)

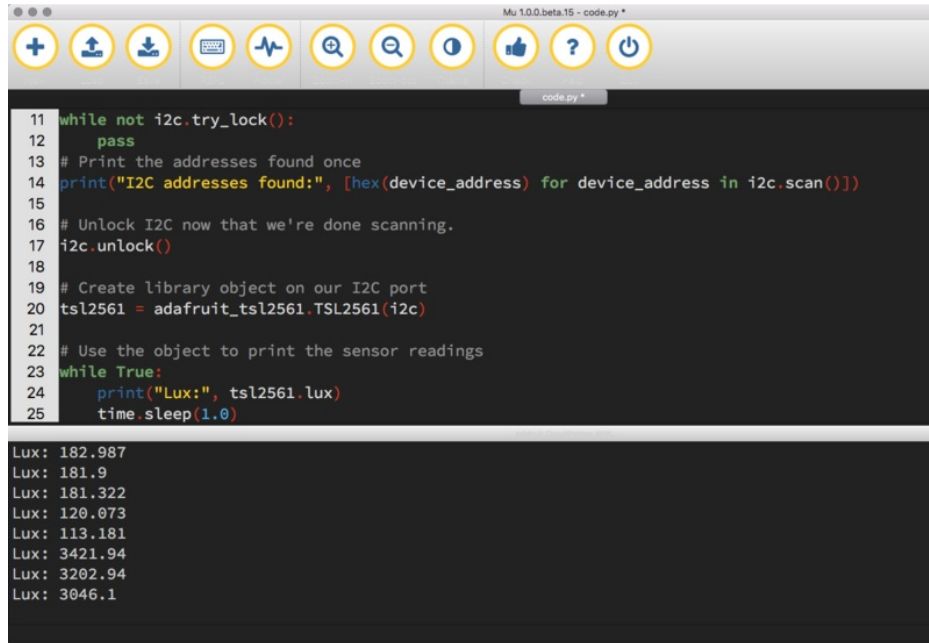
# Use the object to print the sensor readings
while True:
    print("Lux:", tsl2561.lux)
    time.sleep(1.0)

```

This code begins the same way as the scan code. We've included the scan code so you have verification that your sensor is wired up correctly and is detected. It prints the address once. After the scan, we unlock I2C with `i2c_unlock()` so we can use the sensor for data.

We create our sensor object using the sensor library. We call it `tsl2561` and provide it the `i2c` object.

Then we have a simple loop that prints out the lux reading using the sensor object we created. We add a `time.sleep(1.0)`, so it only prints once per second. Connect to the serial console to see the results. Try shining a light on it to see the results change!



```
Mu 1.0.0.beta.15 - code.py *
+ [upload] [download] [terminal] [refresh] [undo] [redo] [help] [power]
code.py *
11 while not i2c.try_lock():
12     pass
13 # Print the addresses found once
14 print("I2C addresses found:", [hex(device_address) for device_address in i2c.scan()])
15
16 # Unlock I2C now that we're done scanning.
17 i2c.unlock()
18
19 # Create library object on our I2C port
20 tsl2561 = adafruit_tsl2561.TSL2561(i2c)
21
22 # Use the object to print the sensor readings
23 while True:
24     print("Lux:", tsl2561.lux)
25     time.sleep(1.0)

Lux: 182.987
Lux: 181.9
Lux: 181.322
Lux: 120.073
Lux: 113.181
Lux: 3421.94
Lux: 3202.94
Lux: 3046.1
```

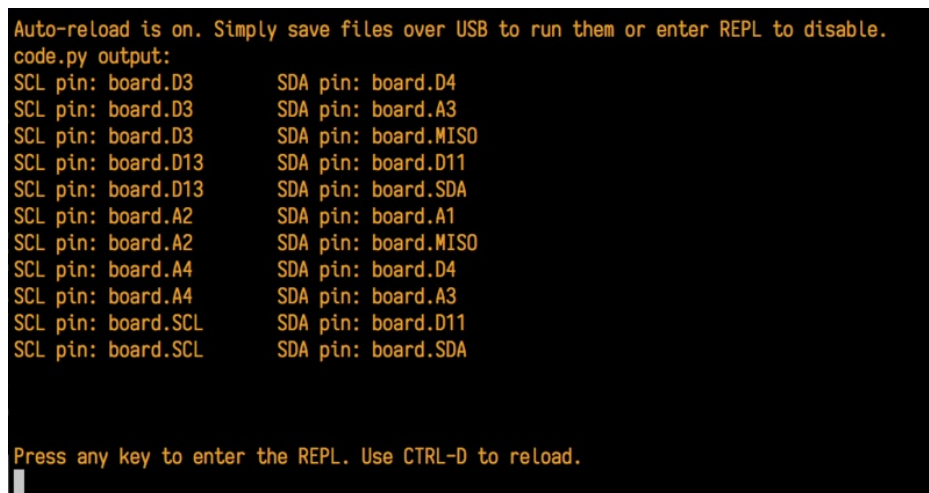
Where's my I2C?

On the SAMD21, we have the flexibility of using a wide range of pins for I2C. Some chips, like the ESP8266 can use *any* pins for I2C, using bitbangio. There's some other chips that may have fixed I2C pin.

The good news is you can use many but not *all* pins. Given the large number of SAMD boards we have, its impossible to guarantee anything other than the labeled 'SDA' and 'SCL'. So, if you want some other setup, or multiple I2C interfaces, how will you find those pins? Easy! We've written a handy script.

All you need to do is copy this file to your board, rename it **code.py**, connect to the serial console and check out the output! The results print out a nice handy list of SCL and SDA pin pairs that you can use.

These are the results from an ItsyBitsy M0 Express. Your output may vary and it might be *very* long. For more details about I2C and SERCOMs, [check out our detailed guide here \(https://adafru.it/Ben\)](https://adafru.it/Ben).



```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
SCL pin: board.D3      SDA pin: board.D4
SCL pin: board.D3      SDA pin: board.A3
SCL pin: board.D3      SDA pin: board.MISO
SCL pin: board.D13     SDA pin: board.D11
SCL pin: board.D13     SDA pin: board.SDA
SCL pin: board.A2      SDA pin: board.A1
SCL pin: board.A2      SDA pin: board.MISO
SCL pin: board.A4      SDA pin: board.D4
SCL pin: board.A4      SDA pin: board.A3
SCL pin: board.SCL     SDA pin: board.D11
SCL pin: board.SCL     SDA pin: board.SDA

Press any key to enter the REPL. Use CTRL-D to reload.
```

```

import board
import busio
from microcontroller import Pin

def is_hardware_I2C(scl, sda):
    try:
        p = busio.I2C(scl, sda)
        p.deinit()
        return True
    except ValueError:
        return False
    except RuntimeError:
        return True

def get_unique_pins():
    exclude = ['NEOPIXEL', 'APA102_MOSI', 'APA102_SCK']
    pins = [pin for pin in [
        getattr(board, p) for p in dir(board) if p not in exclude]
        if isinstance(pin, Pin)]
    unique = []
    for p in pins:
        if p not in unique:
            unique.append(p)
    return unique

for scl_pin in get_unique_pins():
    for sda_pin in get_unique_pins():
        if scl_pin is sda_pin:
            continue
        else:
            if is_hardware_I2C(scl_pin, sda_pin):
                print("SCL pin:", scl_pin, "\t SDA pin:", sda_pin)
            else:
                pass

```

CircuitPython HID Keyboard and Mouse

One of the things we baked into CircuitPython is 'HID' (**H**uman **I**nterface **D**evice) control - that means keyboard and mouse capabilities. This means your CircuitPython board can act like a keyboard device and press key commands, or a mouse and have it move the mouse pointer around and press buttons. This is really handy because even if you cannot adapt your software to work with hardware, there's almost always a keyboard interface - so if you want to have a capacitive touch interface for a game, say, then keyboard emulation can often get you going really fast!

This section walks you through the code to create a keyboard or mouse emulator. First we'll go through an example that uses pins on your board to emulate keyboard input. Then, we will show you how to wire up a joystick to act as a mouse, and cover the code needed to make that happen.

You'll need the **adafruit_hid** library folder if you don't already have it in your `/lib` folder! You can get it from the [CircuitPython Library Bundle \(https://adafruit.it/y8E\)](https://adafruit.it/y8E). If you need help installing the library, check out the [CircuitPython Libraries page \(https://adafruit.it/ABU\)](https://adafruit.it/ABU).

CircuitPython Keyboard Emulator

Copy and paste the code into **code.py** using your favorite editor, and save the file.

```
# CircuitPython demo - Keyboard emulator

import time

import board
import digitalio
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keyboard_layout_us import KeyboardLayoutUS
from adafruit_hid.keycode import Keycode

# A simple neat keyboard demo in CircuitPython

# The pins we'll use, each will have an internal pullup
keypress_pins = [board.A1, board.A2]
# Our array of key objects
key_pin_array = []
# The Keycode sent for each button, will be paired with a control key
keys_pressed = [Keycode.A, "Hello World!\n"]
control_key = Keycode.SHIFT

# The keyboard object!
time.sleep(1) # Sleep for a bit to avoid a race condition on some systems
keyboard = Keyboard()
keyboard_layout = KeyboardLayoutUS(keyboard) # We're in the US :)

# Make all pin objects inputs with pullups
for pin in keypress_pins:
    key_pin = digitalio.DigitalInOut(pin)
    key_pin.direction = digitalio.Direction.INPUT
    key_pin.pull = digitalio.Pull.UP
    key_pin_array.append(key_pin)

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT
```

```

print("Waiting for key pin...")

while True:
    # Check each pin
    for key_pin in key_pin_array:
        if not key_pin.value: # Is it grounded?
            i = key_pin_array.index(key_pin)
            print("Pin #%d is grounded." % i)

            # Turn on the red LED
            led.value = True

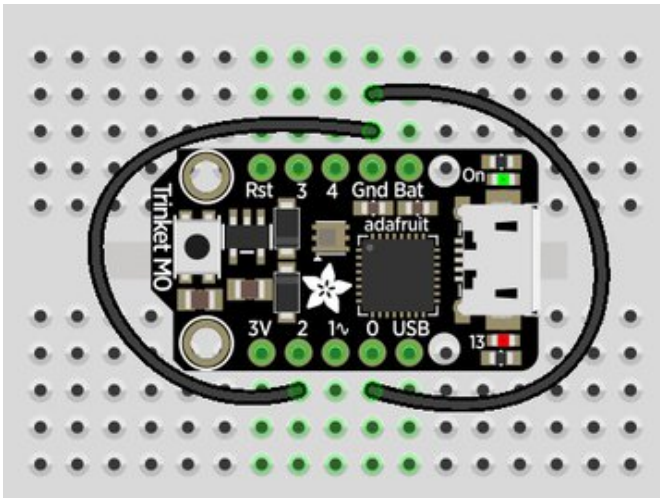
            while not key_pin.value:
                pass # Wait for it to be ungrounded!
            # "Type" the Keycode or string
            key = keys_pressed[i] # Get the corresponding Keycode or string
            if isinstance(key, str): # If it's a string...
                keyboard_layout.write(key) # ...Print the string
            else: # If it's not a string...
                keyboard.press(control_key, key) # "Press"...
                keyboard.release_all() # ..."Release"!

            # Turn off the red LED
            led.value = False

    time.sleep(0.01)

```

Connect pin **A1** or **A2** to ground, using a wire or alligator clip, then disconnect it to send the key press "A" or the string "Hello world!"



This wiring example shows A1 and A2 connected to ground.

Remember, on Trinket, A1 and A2 are labeled 2 and 0! On other boards, you will have A1 and A2 labeled as expected.

Create the Objects and Variables

First, we assign some variables for later use. We create three arrays assigned to variables: `keypress_pins`, `key_pin_array`, and `keys_pressed`. The first is the pins we're going to use. The second is empty because we're going to fill it later. The third is what we would like our "keyboard" to output - in this case the letter "A" and the phrase, "Hello world!". We create our last variable assigned to `control_key` which allows us to later apply the shift key to our keypress. We'll be using two keypresses, but you can have up to six keypresses at once.

Next `keyboard` and `keyboard_layout` objects are created. We only have US right now (if you make other layouts please submit a GitHub pull request!). The `time.sleep(1)` avoids an error that can happen if the program gets run as soon as the board gets plugged in, before the host computer finishes connecting to the board.

Then we take the pins we chose above, and create the pin objects, set the direction and give them each a pullup. Then we apply the pin objects to `key_pin_array` so we can use them later.

Next we set up the little red LED to so we can use it as a status light.

The last thing we do before we start our loop is `print`, "Waiting for key pin..." so you know the code is ready and waiting!

The Main Loop

Inside the loop, we check each pin to see if the state has changed, i.e. you connected the pin to ground. Once it changes, it prints, "Pin # grounded." to let you know the ground state has been detected. Then we turn on the red LED. The code waits for the state to change again, i.e. it waits for you to unground the pin by disconnecting the wire attached to the pin from ground.

Then the code gets the corresponding keys pressed from our array. If you grounded and ungrounded A1, the code retrieves the keypress `a`, if you grounded and ungrounded A2, the code retrieves the string, `"Hello world!"`

If the code finds that it's retrieved a string, it prints the string, using the `keyboard_layout` to determine the keypresses. Otherwise, the code prints the keypress from the `control_key` and the keypress "a", which result in "A". Then it calls `keyboard.release_all()`. You always want to call this soon after a keypress or you'll end up with a stuck key which is really annoying!

Instead of using a wire to ground the pins, you can try wiring up buttons like we did in [CircuitPython Digital In & Out \(https://adafruit.it/Beo\)](https://adafruit.com/blog/2017-08-24-circuitpython-digital-in-out/). Try altering the code to add more pins for more keypress options!

CircuitPython Mouse Emulator

Copy and paste the code into `code.py` using your favorite editor, and save the file.

```
import time

import analogio
import board
import digitalio
from adafruit_hid.mouse import Mouse

mouse = Mouse()

x_axis = analogio.AnalogIn(board.A0)
y_axis = analogio.AnalogIn(board.A1)
select = digitalio.DigitalInOut(board.A2)
select.direction = digitalio.Direction.INPUT
select.pull = digitalio.Pull.UP

pot_min = 0.00
pot_max = 3.29
step = (pot_max - pot_min) / 20.0
```

```

def get_voltage(pin):
    return (pin.value * 3.3) / 65536

def steps(axis):
    """ Maps the potentiometer voltage range to 0-20 """
    return round((axis - pot_min) / step)

while True:
    x = get_voltage(x_axis)
    y = get_voltage(y_axis)

    if select.value is False:
        mouse.click(Mouse.LEFT_BUTTON)
        time.sleep(0.2) # Debounce delay

    if steps(x) > 11.0:
        # print(steps(x))
        mouse.move(x=1)
    if steps(x) < 9.0:
        # print(steps(x))
        mouse.move(x=-1)

    if steps(x) > 19.0:
        # print(steps(x))
        mouse.move(x=8)
    if steps(x) < 1.0:
        # print(steps(x))
        mouse.move(x=-8)

    if steps(y) > 11.0:
        # print(steps(y))
        mouse.move(y=-1)
    if steps(y) < 9.0:
        # print(steps(y))
        mouse.move(y=1)

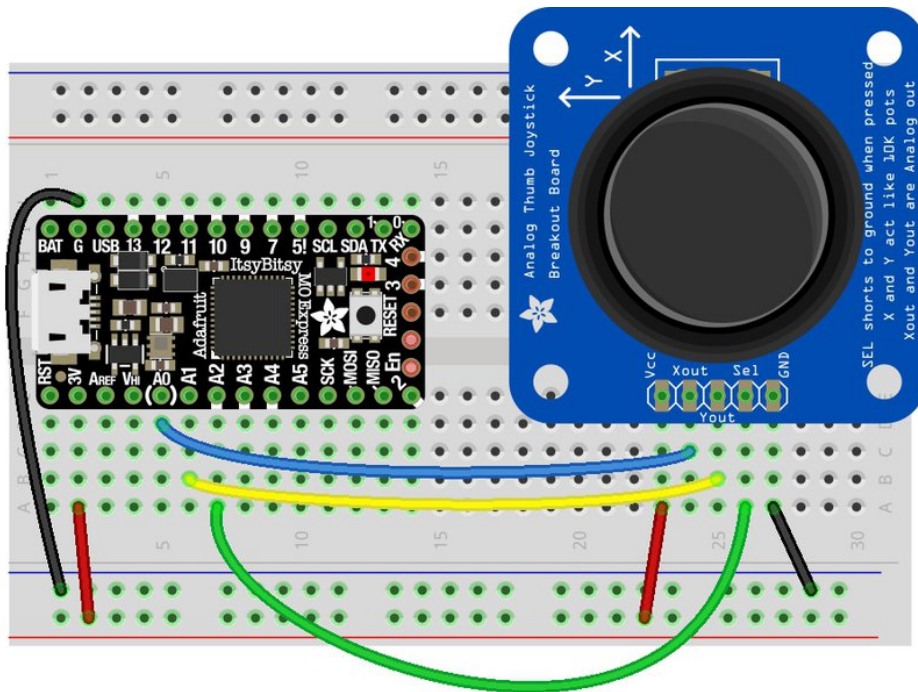
    if steps(y) > 19.0:
        # print(steps(y))
        mouse.move(y=-8)
    if steps(y) < 1.0:
        # print(steps(y))
        mouse.move(y=8)

```

For this example, we've wired up a 2-axis thumb joystick with a select button. We use this to emulate the mouse movement and the mouse left-button click. To wire up this joystick:

- Connect **VCC** on the joystick to the **3V** on your board. Connect **ground** to **ground**.
- Connect **Xout** on the joystick to pin **A0** on your board.
- Connect **Yout** on the joystick to pin **A1** on your board.
- Connect **Sel** on the joystick to pin **A2** on your board.

Remember, Trinket's pins are labeled differently. Check the [Trinket Pinouts page \(https://adafruit.it/AMd\)](https://adafruit.it/AMd) to verify your wiring.



fritzing

To use this demo, simply move the joystick around. The mouse will move slowly if you move the joystick a little off center, and more quickly if you move it as far as it goes. Press down on the joystick to click the mouse. Awesome! Now let's take a look at the code.

Create the Objects and Variables

First we create the mouse object.

Next, we set `x_axis` and `y_axis` to pins `A0` and `A1`. Then we set `select` to `A2`, set it as input and give it a pullup.

The x and y axis on the joystick act like 2 potentiometers. We'll be using them just like we did in [CircuitPython Analog In \(https://adafru.it/Bep\)](https://adafru.it/Bep). We set `pot_min` and `pot_max` to be the minimum and maximum voltage read from the potentiometers. We assign `step = (pot_max - pot_min) / 20.0` to use in a helper function.

CircuitPython HID Mouse Helpers

First we have the `get_voltage()` helper so we can get the correct readings from the potentiometers. Look familiar? We [learned about it in Analog In \(https://adafru.it/Bep\)](https://adafru.it/Bep).

Second, we have `steps(axis)`. To use it, you provide it with the axis you're reading. This is where we're going to use the `step` variable we assigned earlier. The potentiometer range is 0-3.29. This is a small range. It's even smaller with the joystick because the joystick sits at the center of this range, 1.66, and the + and - of each axis is above and below this number. Since we need to have thresholds in our code, we're going to map that range of 0-3.29 to while numbers between 0-20.0 using this helper function. That way we can simplify our code and use larger ranges for our thresholds instead of trying to figure out tiny decimal number changes.

Main Loop

First we assign `x` and `y` to read the voltages from `x_axis` and `y_axis`.

Next, we check to see when the state of the select button is `False`. It defaults to `True` when it is not pressed, so if the state is `False`, the button has been pressed. When it's pressed, it sends the command to click the left mouse button. The `time.sleep(0.2)` prevents it from reading multiple clicks when you've only clicked once.

Then we use the `steps()` function to set our mouse movement. There are two sets of two `if` statements for each axis. Remember that `10` is the center step, as we've mapped the range `0-20`. The first set for each axis says if the joystick moves 1 step off center (left or right for the x axis and up or down for the y axis), to move the mouse the appropriate direction by 1 unit. The second set for each axis says if the joystick is moved to the lowest or highest step for each axis, to move the mouse the appropriate direction by 8 units. That way you have the option to move the mouse slowly or quickly!

To see what `step` the joystick is at when you're moving it, uncomment the `print` statements by removing the `#` from the lines that look like `# print(steps(x))`, and connecting to the serial console to see the output. Consider only uncommenting one set at a time, or you end up with a huge amount of information scrolling very quickly, which can be difficult to read!



For more detail check out the documentation at <https://circuitpython.readthedocs.io/projects/hid/en/latest/>

CircuitPython CPU Temp

There is a CPU temperature sensor built into every ATSAM21 chip. CircuitPython makes it really simple to read the data from this sensor. This works on the M0, M0 Express and Circuit Playground Express boards, because it's built into the microcontroller used for these boards. It does not work on the ESP8266 as this uses a different chip.

The data is read using two simple commands. We're going to enter them in the REPL. Plug in your board, [connect to the serial console \(https://adafru.it/Bec\)](https://adafru.it/Bec), and [enter the REPL \(https://adafru.it/Awz\)](https://adafru.it/Awz). Then, enter the following commands into the REPL:

```
import microcontroller
microcontroller.cpu.temperature
```

That's it! You've printed the temperature in Celsius to the REPL. Note that it's not exactly the ambient temperature and it's not super precise. But it's close!

```
Adafruit CircuitPython 2.2.4 on 2018-03-07; Adafruit Metro M0 Express with samd21g18
>>> import microcontroller
>>> microcontroller.cpu.temperature
21.8071
>>> █
```

If you'd like to print it out in Fahrenheit, use this simple formula: Celsius * (9/5) + 32. It's super easy to do math using CircuitPython. Check it out!

```
>>> microcontroller.cpu.temperature * (9 / 5) + 32
70.8655
>>> █
```

CircuitPython Storage

CircuitPython boards show up as a USB drive, allowing you to edit code directly on the board. You've been doing this for a while. By now, maybe you've wondered, "Can I write data *from CircuitPython* to the storage drive to act as a datalogger?" The answer is **yes!**

However, it is a little tricky. You need to add some special code to **boot.py**, not just **code.py**. That's because you have to set the filesystem to be read-only when you need to edit code to the disk from your computer, and set it to writeable when you want the CircuitPython core to be able to write.



You can only have either your computer edit the CIRCUITPY drive files, or CircuitPython. You cannot have both write to the drive at the same time. (Bad Things Will Happen so we do not allow you to do it!)

The following is your new **boot.py**. Copy and paste the code into **boot.py** using your favorite editor. You may need to create a new file.

```
import board
import digitalio
import storage

# For Gemma M0, Trinket M0, Metro M0/M4 Express, ItsyBitsy M0/M4 Express
switch = digitalio.DigitalInOut(board.D2)
# switch = digitalio.DigitalInOut(board.D5) # For Feather M0/M4 Express
# switch = digitalio.DigitalInOut(board.D7) # For Circuit Playground Express
switch.direction = digitalio.Direction.INPUT
switch.pull = digitalio.Pull.UP

# If the switch pin is connected to ground CircuitPython can write to the drive
storage.remount("/", switch.value)
```

For **Gemma M0**, **Trinket M0**, **Metro M0 Express**, **Metro M4 Express**, **ItsyBitsy M0 Express** and **ItsyBitsy M4 Express**, no changes to the initial code are needed.

For **Feather M0 Express** and **Feather M4 Express**, comment out `switch = digitalio.DigitalInOut(board.D2)`, and uncomment `switch = digitalio.DigitalInOut(board.D5)`.

For **Circuit Playground Express**, comment out `switch = digitalio.DigitalInOut(board.D2)`, and uncomment `switch = digitalio.DigitalInOut(board.D7)`.



Remember: To "comment out" a line, put a # and a space before it. To "uncomment" a line, remove the # + space from the beginning of the line.

The following is your new **code.py**. Copy and paste the code into **code.py** using your favorite editor.

```

import time

import board
import digitalio
import microcontroller

led = digitalio.DigitalInOut(board.D13)
led.switch_to_output()

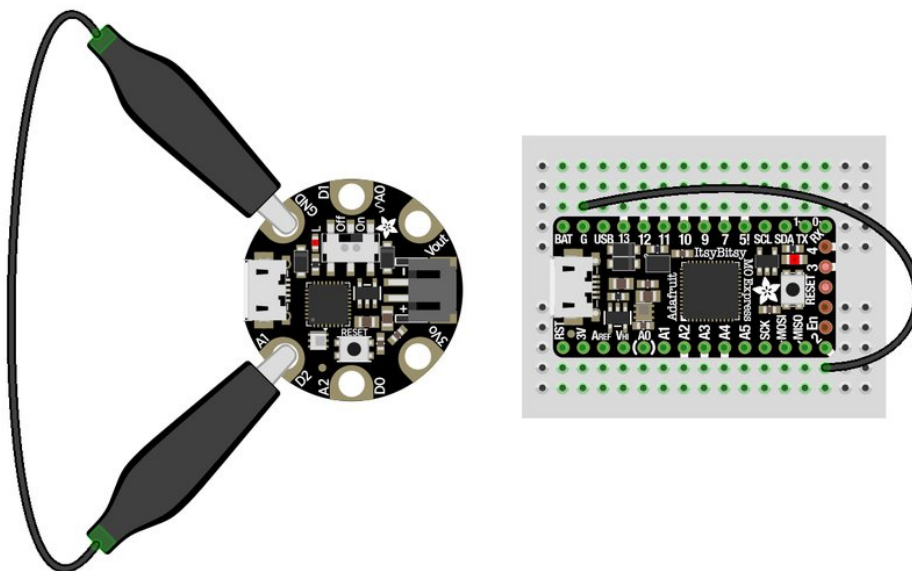
try:
    with open("/temperature.txt", "a") as fp:
        while True:
            temp = microcontroller.cpu.temperature
            # do the C-to-F conversion here if you would like
            fp.write('{0:f}\n'.format(temp))
            fp.flush()
            led.value = not led.value
            time.sleep(1)
except OSError as e:
    delay = 0.5
    if e.args[0] == 28:
        delay = 0.25
    while True:
        led.value = not led.value
        time.sleep(delay)

```

Logging the Temperature

The way `boot.py` works is by checking to see if the pin you specified in the switch setup in your code is connected to a ground pin. If it is, it changes the read-write state of the file system, so the CircuitPython core can begin logging the temperature to the board.

For help finding the correct pins, see the wiring diagrams and information in the [Find the Pins](https://adafruit.it/Bes) section of the [CircuitPython Digital In & Out guide](https://adafruit.it/Bes). Instead of wiring up a switch, however, you'll be connecting the pin directly to ground with alligator clips or jumper wires.



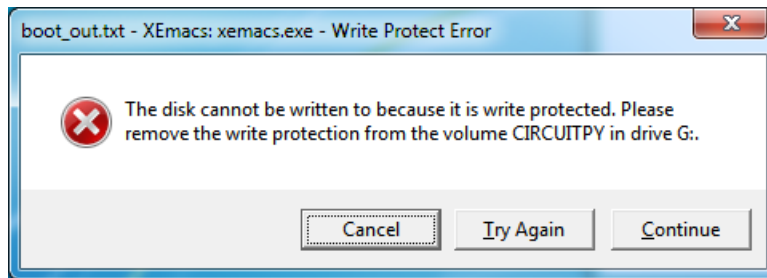
fritzing



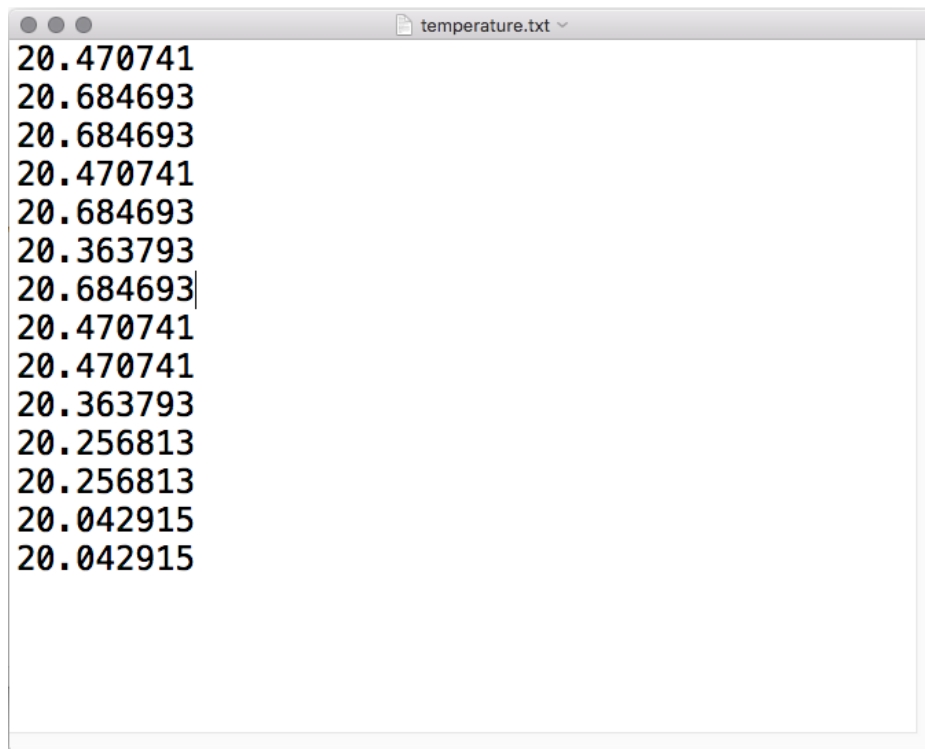
boot.py only runs on first boot of the device, not if you re-load the serial console with ctrl+D or if you save a file. You must EJECT the USB drive, then physically press the reset button!

Once you copied the files to your board, eject it and unplug it from your computer. If you're using your Circuit Playground Express, all you have to do is make sure the switch is to the right. Otherwise, use alligator clips or jumper wires to connect the chosen pin to ground. Then, plug your board back into your computer.

You will not be able to edit code on your CIRCUITPY drive anymore!



The red LED should blink once a second and you will see a new `temperature.txt` file on CIRCUITPY.



This file gets updated once per second, but you won't see data come in live. Instead, when you're ready to grab the data, eject and unplug your board. For CPX, move the switch to the left, otherwise remove the wire connecting the pin to ground. Now it will be possible for you to write to the filesystem from your computer again, but it will not be logging data.

We have a more detailed guide on this project available here: [CPU Temperature Logging with CircuitPython. \(https://adafru.it/zuF\)](https://adafru.it/zuF) If you'd like more details, check it out!

CircuitPython Expectations



As we continue to develop CircuitPython and create new releases, we will stop supporting older releases. If you are running CircuitPython 2.x, you need to update to 3.x. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update to CircuitPython 3.x and then download the 3.x bundle.

Always Run the Latest Version of CircuitPython and Libraries

As we continue to develop CircuitPython and create new releases, we will stop supporting older releases. **If you are running CircuitPython 2.x, you need to [update to 3.x \(https://adafru.it/Amd\)](https://adafru.it/Amd).**

You need to download the CircuitPython Library Bundle that matches your version of CircuitPython. **Please update to CircuitPython 3.x and then [download the 3.x bundle \(https://adafru.it/ABU\)](https://adafru.it/ABU).**

We will soon stop providing the 2.x bundle as an automatically created download on the Adafruit CircuitPython Bundle repo. If you must continue to use 2.x, you can still download the 2.x version of [mpy-cross](#) from the 2.x release of CircuitPython on the CircuitPython repo and create your own 2.x compatible .mpy library files. **However, it is best to update to 3.x for both CircuitPython and the library bundle.**

Switching Between CircuitPython and Arduino

Many of the CircuitPython boards also run Arduino. But how do you switch between the two? Switching between CircuitPython and Arduino is easy.

If you're currently running Arduino and would like to start using CircuitPython, follow the steps found in [Welcome to CircuitPython: Installing CircuitPython \(https://adafru.it/Amd\)](https://adafru.it/Amd).

If you're currently running CircuitPython and would like to start using Arduino, plug in your board, and then load your Arduino sketch. If there are any issues, you can double tap the reset button to get into the bootloader and then try loading your sketch. Always backup any files you're using with CircuitPython that you want to save as they could be deleted.

That's it! It's super simple to switch between the two.

The Difference Between Express And Non-Express Boards

We often reference "Express" and "Non-Express" boards when discussing CircuitPython. What does this mean?

Express refers to the inclusion of an extra 2MB flash chip on the board that provides you with extra space for CircuitPython and your code. This means that we're able to include more functionality in CircuitPython and you're able to do more with your code on an Express board than you would on a non-Express board.

Express boards include Circuit Playground Express, ItsyBitsy M0 Express, Feather M0 Express, Metro M0 Express and Metro M4 Express.

Non-Express boards include Trinket M0, Gemma M0, Feather M0 Basic, and other non-Express Feather M0 variants.

Non-Express Boards: Gemma and Trinket

CircuitPython runs nicely on the Gemma M0 or Trinket M0 but there are some constraints

Small Disk Space

Since we use the internal flash for disk, and that's shared with runtime code, it's limited! Only about 50KB of space.

No Audio or NVM

Part of giving up that FLASH for disk means we couldn't fit everything in. There is, at this time, no support for hardware audio playback or NVM 'eeprom'. Modules `audioio` and `bitbangio` are not included. For that support, check out the Circuit Playground Express or other Express boards.

However, I2C, UART, capacitive touch, NeoPixel, DotStar, PWM, analog in and out, digital IO, logging storage, and HID do work! Check the CircuitPython Essentials for examples of all of these.

Differences Between CircuitPython and MicroPython

For the differences between CircuitPython and MicroPython, check out the [CircuitPython documentation \(https://adafru.it/Bvz\)](https://adafru.it/Bvz).

Differences Between CircuitPython and Python

Python (also known as CPython) is the language that MicroPython and CircuitPython are based on. There are many similarities, but there are also many differences. This is a list of a few of the differences.

Python Libraries

Python is advertised as having "batteries included", meaning that many standard libraries are included. Unfortunately, for space reasons, many Python libraries are not available. So for instance while we wish you could `import numpy`, `numpy` isn't available. So you may have to port some code over yourself!

Integers in CircuitPython

On the non-Express boards, integers can only be up to 31 bits long. Integers of unlimited size are not supported. The largest positive integer that can be represented is $2^{30}-1$, 1073741823, and the most negative integer possible is -2^{30} , -1073741824.

As of CircuitPython 3.0, Express boards have arbitrarily long integers as in Python.

Floating Point Numbers and Digits of Precision for Floats in CircuitPython

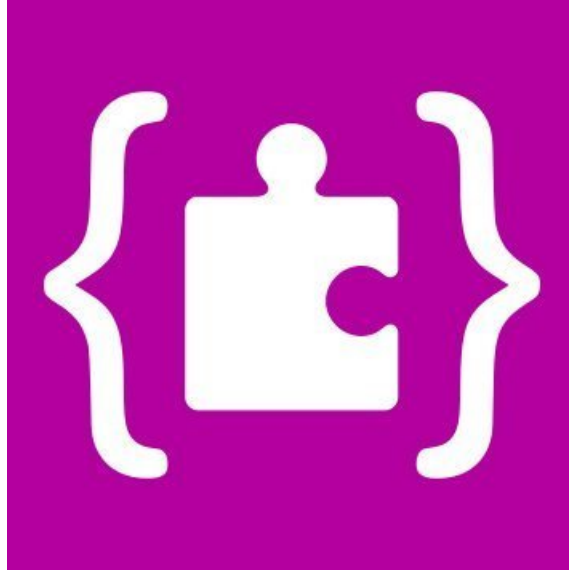
Floating point numbers are single precision in CircuitPython (not double precision as in Python). The largest floating point magnitude that can be represented is about $\pm 3.4e38$. The smallest magnitude that can be represented with full accuracy is about $\pm 1.7e-38$, though numbers as small as $\pm 5.6e-45$ can be represented with reduced accuracy.

CircuitPython's floats have 8 bits of exponent and 22 bits of mantissa (not 24 like regular single precision floating point), which is about five or six decimal digits of precision.

Differences between MicroPython and Python

For a more detailed list of the differences between CircuitPython and Python, you can look at the MicroPython documentation. [We keep up with MicroPython stable releases, so check out the core 'differences' they document here. \(https://adafru.it/zwA\)](https://adafru.it/zwA)

MakeCode

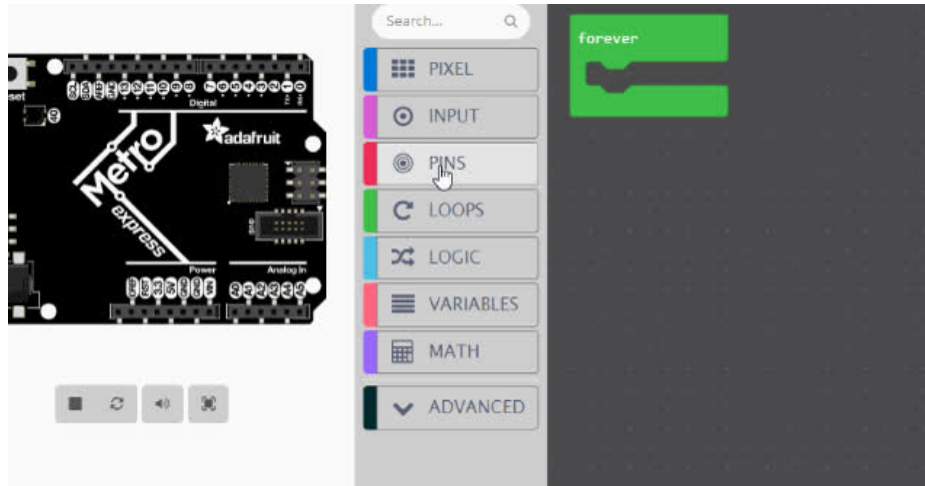


Microsoft MakeCode has been augmented to support more than the Adafruit Circuit Playground Express.

Using [maker.makecode.com](https://adafru.it/C9N) (<https://adafru.it/C9N>), you can use other Adafruit microcontrollers, breadboards and other components!

See the following pages for more information.

What is MakeCode Maker?



MakeCode Maker, <https://maker.makecode.com>, is a web-based code editor for physical computing. It provides a block editor, similar to Scratch or Code.org, and also a JavaScript editor for more advanced users.

Some of the key features of MakeCode are:

- **web based editor:** nothing to install
- **cross platform:** works in most modern browsers from tiny phone to giant touch screens
- **compilation in the browser:** the compiler runs in your browser, it's fast and works offline
- **blocks + JavaScript:** drag and drop blocks or type JavaScript, MakeCode let's you go back and forth between the two.
- **works offline:** once you've loaded the editor, it stays cached in your browser.
- **event based runtime:** easily respond to button clicks, shake gestures and more

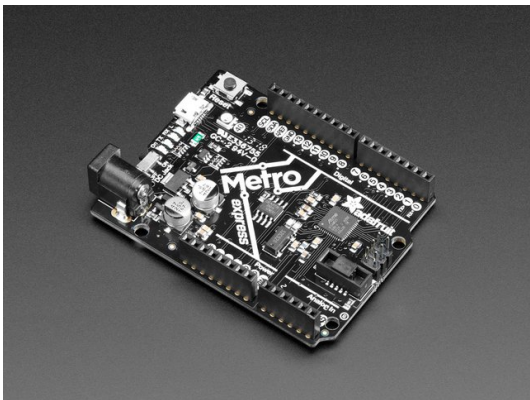
□ How is it related to makecode.adafruit.com ?

makecode.adafruit.com and maker.makecode.com are editors built using the [MakeCode](https://makecode.com) project. In both editors, one can use drag-and-drop blocks or JavaScript to program micro-controllers.

- makecode.adafruit.com specifically applies to the **Adafruit Circuit Playground Express only**
- maker.makecode.com aims at supporting the **Adafruit Express** boards (and more boards from different manufacturers), with an emphasis on breadboarding support.

📄 Is it open source?

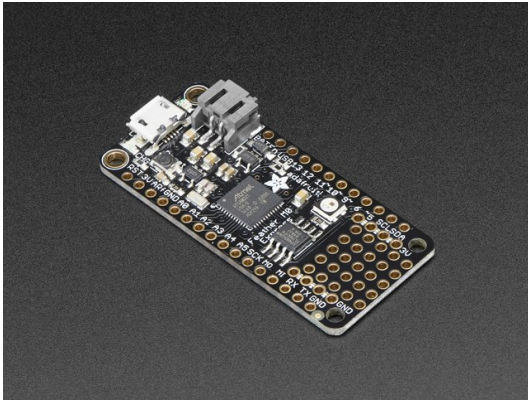
Yes, Maker is open source under MIT at <https://github.com/Microsoft/pxt-maker>.



Adafruit METRO M0 Express - designed for CircuitPython

\$24.95
IN STOCK

ADD TO CART



Adafruit Feather M0 Express - Designed for CircuitPython

\$19.95
IN STOCK

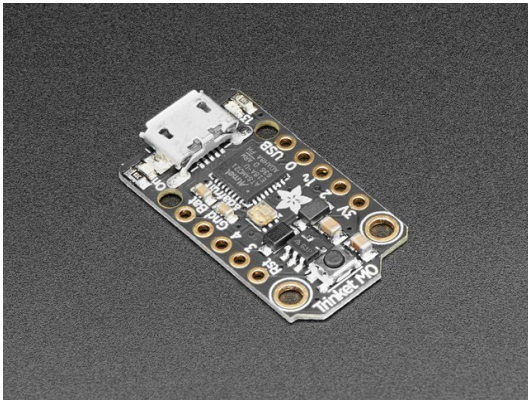
ADD TO CART

Your browser does not support the video tag.

Adafruit GEMMA M0 - Miniature wearable electronic platform

\$9.95
IN STOCK

ADD TO CART



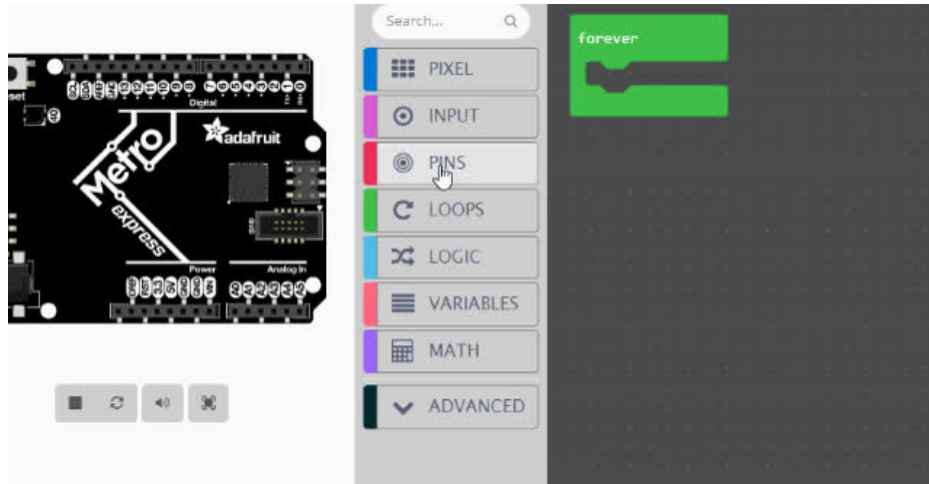
Adafruit Trinket M0 - for use with CircuitPython & Arduino IDE

\$8.95
IN STOCK

ADD TO CART

Editing Blocks

The block editor is the easiest way to get started with MakeCode Maker. You can drag and drop blocks from the category list. Each time you make a change to the blocks, the simulator will automatically restart and run the code. You can test your program in the browser! The simulator will also generate the wiring for your breadboard for simple programs.



On the maker home screen, click on "New Project", then select which board you want to use (you can change board later too).

Blinky!

The animation above shows to use the blocks to create a program that blinks an LED.

Creating a blink effect is done by setting the pin HIGH, **pause** for a little, then set the pin LOW, pause for a little, then repeat **forever**.

- **forever** runs blocks in a loop with a 20ms pause in between (it is similar to Arduino**loop**).
- **digital write pin** sets the pin to high or low
- **pause** blocks the current thread for 100ms. If other events or forever loops are running, they have the opportunity to run in parallel.

"forever" runs the nested code every 20ms

```
forever
  digital write pin D13 to LOW
  pause 500 ms
  digital write pin D13 to HIGH
  pause 500 ms
```

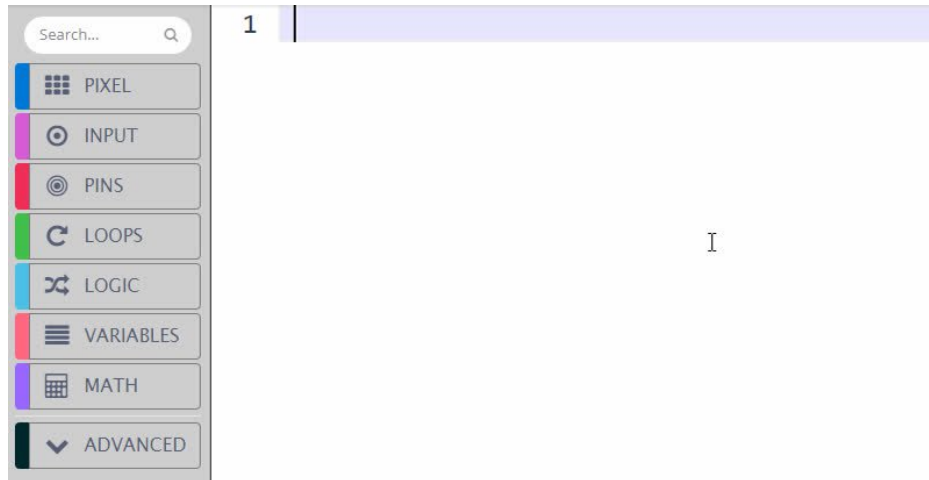
set D13 to HIGH

set D13 to LOW

pause is non blocking, it allows other code to run

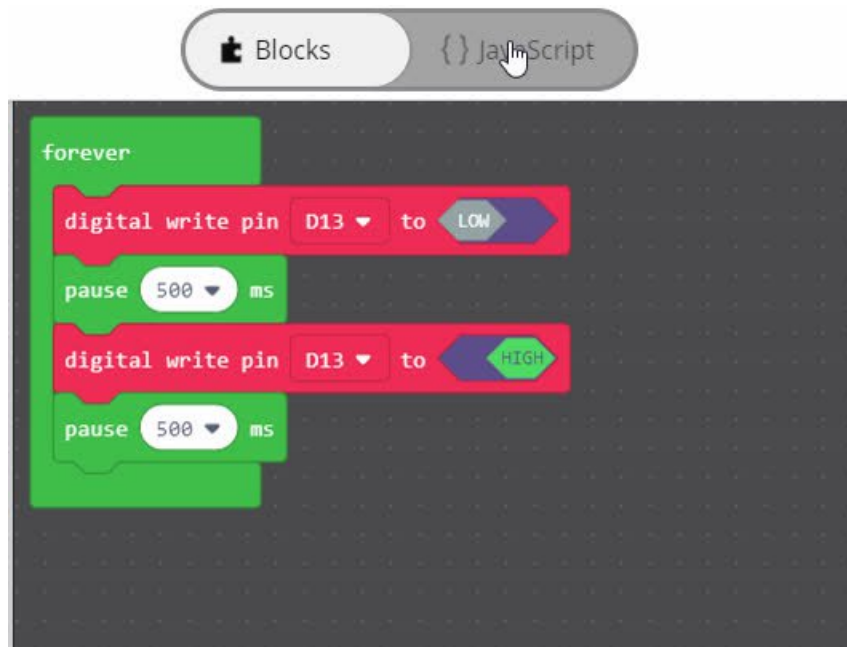
Editing JavaScript

MakeCode allows you to author your programs in a flavor of JavaScript optimized for micro-controllers. The code editor comes with error highlighting, auto-completion and other goodies. It is the same code editor that powers Visual Studio Code.



Blocks to JavaScript

Click on the "Blocks / JavaScript" toggle on top of the editor to enter the JavaScript mode. Your blocks will automatically be converted to JavaScript.



Downloading and Flashing

Getting your code into your device is very easy with MakeCode. You do not need to install any software on your machine and the process takes two steps:

- Step 1: Connect your board via USB
- Step 2: **Compile and Download** the .uf2 file into your board drive

We are going to go through these two steps in detail.

Step 1: Connect your board via USB

Connect your board to your computer via a USB cable. You should see a **MAKECODE** drive appear in your file explorer/finder. If your board is in bootloader mode, you will see drive names like **METROBOOT** or **GEMMABOOT**. We will call those **boardnameBOOT**.

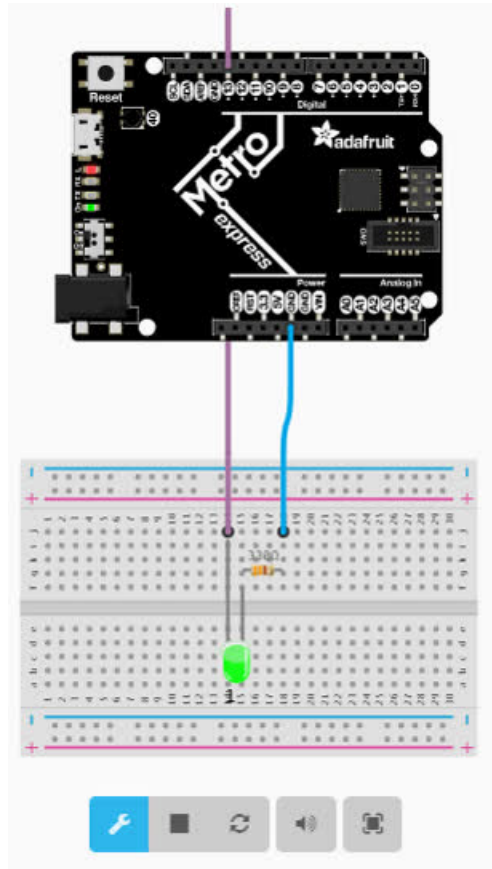


If it is your first time running MakeCode or if you have previously installed Arduino or CircuitPython, you may need to double press the reset button to get your board into bootloader mode.

Step 2: Test your code in the simulator

Let's first verify that our code compiles properly in MakeCode.

MakeCode has a built-in simulator that re-loads and re-runs code when restarted. This is an easy way to both ensure that our code compiles and simulate it before moving it onto the board. The refresh button re-loads the simulator with your latest version of block code.



□ If you receive a "we could not run this project" error, please check over your code for errors.

Step 3: Download and flash your code

If your board is working in the simulator, it's time to download it to your actual board! Click the **Download** button. It will generate a .uf2 file and download it to your computer. [UF2 \(https://adafru.it/vPE\)](https://adafru.it/vPE) is a file format designed by Microsoft to flash microcontrollers over USB.

General Steps to copy over your program (not specific to any Operating system)

- * Ensure your board is connected via USB.
- * Find the .uf2 file generated by MakeCode in your file explorer. Copy it to the **MAKECODE** or **boardnameBOOT** volume.
- * The status LED on the board will blink while the file is transferring. Once it's done transferring your file, the board will automatically reset and start running your code (just like in the simulator!)

□ On a Mac, you can safely ignore the "Disk Not Ejected Properly" notification that may appear after copying your .uf2 file.

Saving and Sharing

Extracting your code from the board

The .uf2 file you created by clicking on the Compile button in MakeCode also contains the source code of your program!

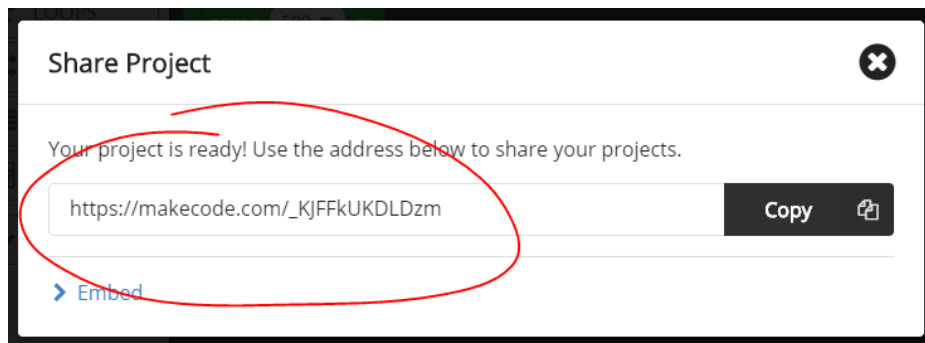
You can open this file in MakeCode by dragging and dropping it into the browser to edit it.

You can also find the current .uf2 file running on the **MAKECODE** or **boardnameBOOT** drive.

Sharing

You can share your code by clicking on the share button. After confirmation, MakeCode will create a short unique URL for your code. Anyone with that URL will be able to reload the code.

These URLs can also be used to embed the editor your blog or web pages! Just copy paste the URL in your text editor and (if it supports oEmbed) it will automatically load it in your page.



Custom Extensions

MakeCode allows to package and share code as **Extensions**. **Extensions** are stored as [GitHub](#) repositories and can be edited directly from the MakeCode editor.

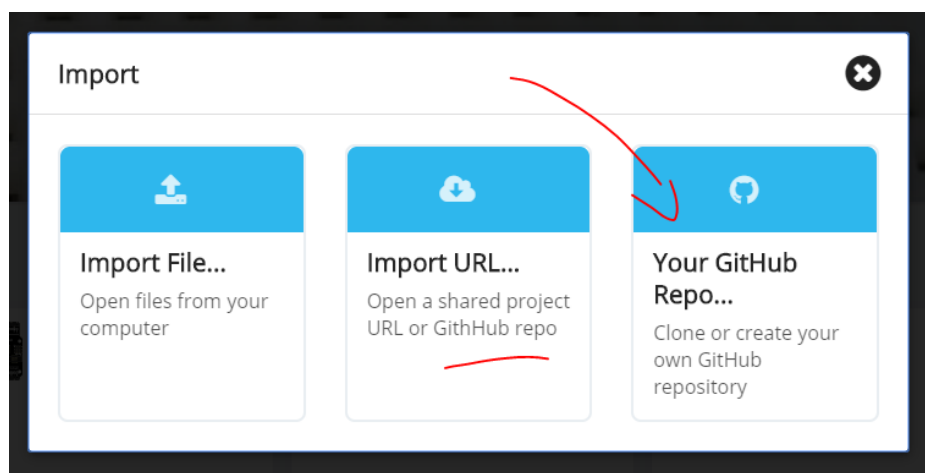
Account setup

First, you need a [GitHub account](#) if you don't have one yet. GitHub is the largest host of source code in the world, with over 30 million users.

Once you have your account, you'll need to tie the MakeCode web app to your account. To do that, open any project in <https://maker.makecode.com>, go to the **Gear Wheel** menu on top, and select **Extensions**. At the bottom, there should be a link to log in to GitHub. A dialog will appear asking you to generate a GitHub token. Follow the instructions and paste the token into the dialog.



Once you've logged in, go back to the home screen. Now, the dialog that comes up after you press the **Import** button will have an additional option to list your GitHub repositories or create a new one. Additionally, the **Import URL** option will now support <https://github.com/...> URLs, which is useful if you can't find your repository in the list (especially organizational repos), or as way to search the list faster using a copy/paste of the URL.



If you import a completely empty repo, or create a fresh one, MakeCode will automatically initialize it with [pxt.json](#) and other supporting files. If you import a non-empty repo without the [pxt.json](#) file, you will be asked if you want it initialized. Note that this might overwrite your existing files.

Create GitHub repo

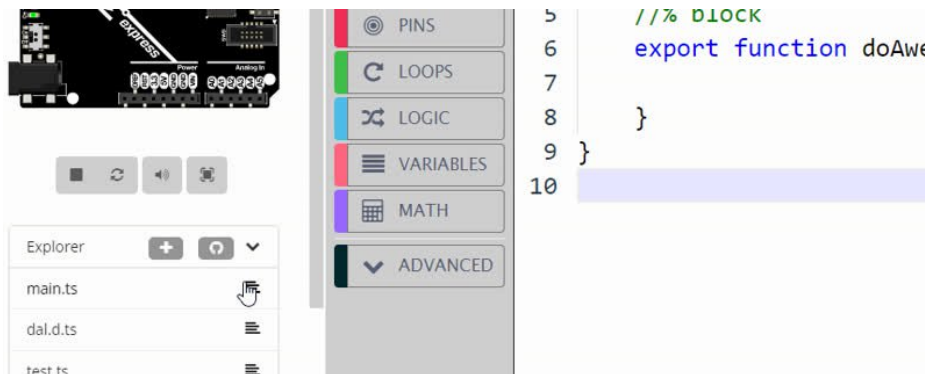
Repo name.

Repo description.

Go ahead! ✓ Cancel ✕

Commit and push

Once you have your repo set up, edit files as usual. Whenever you get to a stable state, or just every now and then to keep history and insure against losing your work, push the changes to GitHub. This is done with a little GitHub sync button on top of the **Explorer**. The button will check if there are any pending changes to check in. If there are, it will create a commit, pull the latest changes from GitHub, merge or fast-forward the commit if needed, and push the results to GitHub.

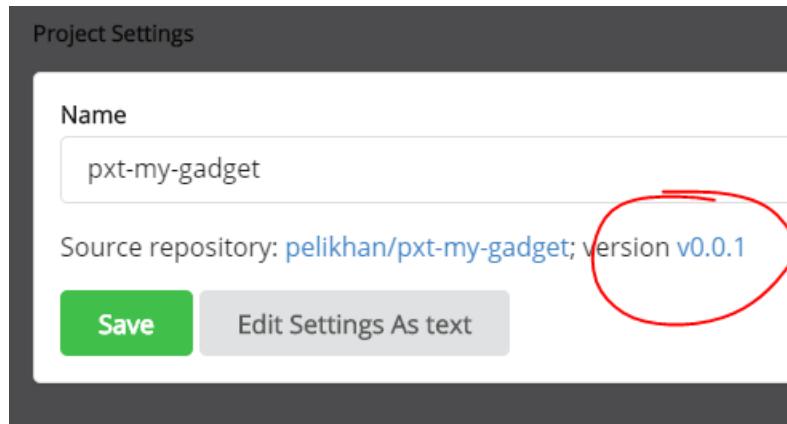


If there are changes, you will be asked for a commit message. Try to write something meaningful, like `Fixed temperature reading in sub-freezing conditions` or `Added mpyensor.readTemperature() function`.

When describing changes, you are also given an option to **bump** the version number. This is a signal that the version you're pushing is stable and the users should upgrade to it. When your package is first referenced, the latest bumped version is used. Similarly, if there is a newer bumped version, a little upgrade button will appear next to the package. Commits without bump are generally not accessible to most users, so they are mostly for you to keep track of things.

We do not really distinguish between a commit, push, and pull - it all happens at once in the sync operation.

You can view a history of changes by following the version number link on the **Project Settings** page.



There is also another button next to the GitHub sync - you can use it to add new files to the project. This is mostly to help keep the project organized. For our TypeScript compiler it doesn't matter if you use one big file or a bunch of smaller ones.

Conflicts

It's possible that multiple people are editing the same package at the same time causing edit conflicts. This is similar to the situation where the same person edits the package using several computers, browsers, or web sites. In the conflict description below, for simplicity, we'll just concentrate on the case of multiple people working on the same package.

Typically, two people would sync a GitHub package at the same version, and then they both edit it. The first person pushes the changes successfully. When MakeCode tries to push the changes from the second person, it will notice that these are changes against a non-current version. It will create a commit based on the previous version and try to use the standard git merge (run server-side by GitHub). This usually succeeds if the two people edited different files, or at least different parts of the file - you end up with both sets of changes logically combined. There is no user interaction required in that case.

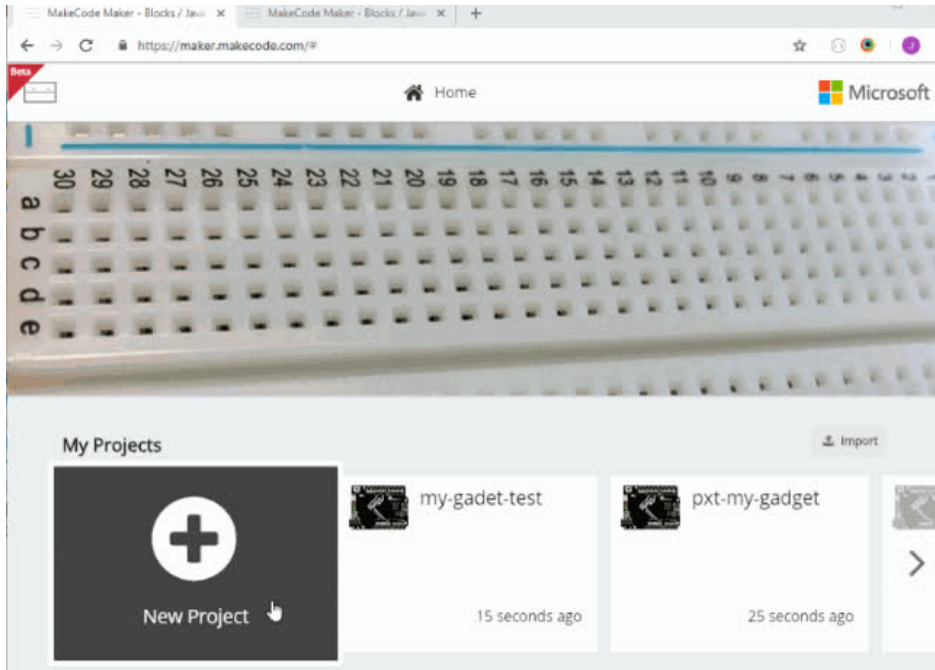
If the automatic merge fails, MakeCode will create a new branch, push the commit there, and then create a pull request (PR) on GitHub. The dialog that appears after this happens will let you go to the GitHub web site and resolve the conflicts. Before you resolve conflicts and merge the PR, the `master` branch will not have your changes (it will have changes from the other person, who managed to commit first). After creating the PR, MakeCode moves your local version to the `master` branch (without your changes), but don't despair they are not lost! Just resolve the conflict in GitHub and sync to get all changes back. MakeCode will also sync automatically when you close the PR dialog (presumably, after you resolved the conflict in another tab).

Testing your package

To test blocks in your package, press the **New Project** button on the home screen and go to the **Extensions** dialog. It will list all your GitHub projects as available for addition. Select your package and see what the blocks look like.

You can have one browser tab open with that test project, and another one with the package. When you switch between them, they reload automatically.

For testing TypeScript APIs you don't need a separate project, and instead can use the `test.ts` file in the package itself. It is only used when you run the package directly, not when you add it to a project. You can put TypeScript test code in there.



UF2 Bootloader Details



This is an information page for advanced users who are curious how we get code from your computer into your Express board!

Adafruit SAMD21 (M0) and SAMD51 (M4) boards feature an improved bootloader that makes it easier than ever to flash different code onto the microcontroller. This bootloader makes it easy to switch between Microsoft MakeCode, CircuitPython and Arduino.

Instead of needing drivers or a separate program for flashing (say, `bossac`, `jlink` or `avrdude`), one can simply *drag a file onto a removable drive*.

The format of the file is a little special. Due to 'operating system woes' you cannot just drag a binary or hex file (trust us, we tried it, it isn't cross-platform compatible). Instead, the format of the file has extra information to help the bootloader know where the data goes. The format is called UF2 (USB Flashing Format). Microsoft MakeCode generates UF2s for flashing and CircuitPython releases are also available as UF2. [You can also create your own UF2s from binary files using uf2tool, available here. \(https://adafru.it/vPE\)](https://adafru.it/vPE)

The bootloader is *also BOSSA compatible*, so it can be used with the Arduino IDE which expects a BOSSA bootloader on ATSAMd-based boards

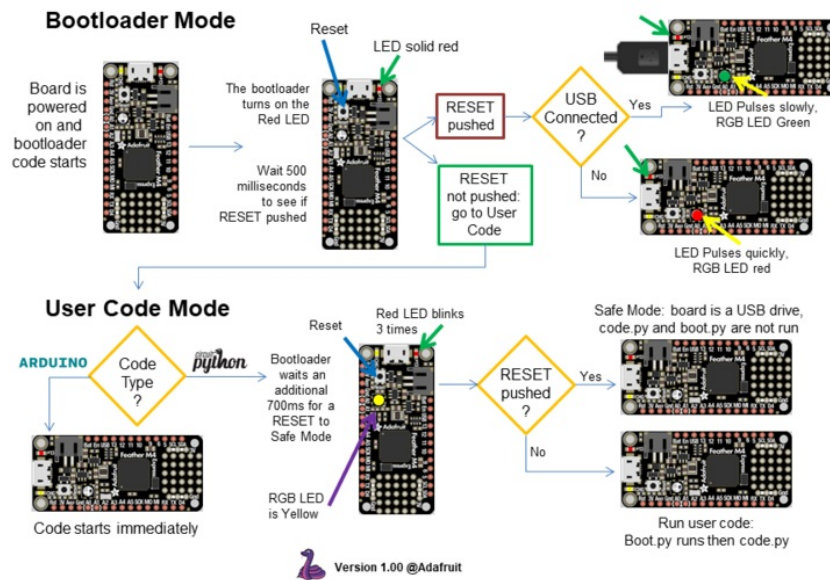
For more information about UF2, [you can read a bunch more at the MakeCode blog \(https://adafru.it/w5A\)](https://adafru.it/w5A), then [check out the UF2 file format specification. \(https://adafru.it/vPE\)](https://adafru.it/vPE)

Visit [Adafruit's fork of the Microsoft UF2-samd bootloader GitHub repository \(https://adafru.it/Beu\)](https://adafru.it/Beu) for source code and releases of pre-built bootloaders on [CircuitPython.org \(https://adafru.it/Em8\)](https://adafru.it/Em8).



The bootloader is not needed when changing your CircuitPython code. Its only needed when upgrading the CircuitPython core or changing between CircuitPython, Arduino and Microsoft MakeCode.

The CircuitPython Boot Sequence

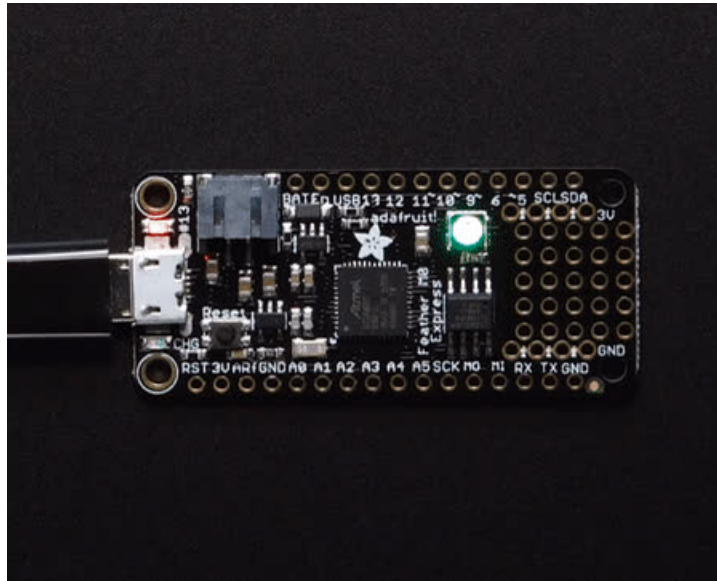


Entering Bootloader Mode

The first step to loading new code onto your board is triggering the bootloader. It is easily done by double tapping the reset button. Once the bootloader is active you will see the small red LED fade in and out and a new drive will appear on your computer with a name ending in **BOOT**. For example, feathers show up as **FEATHERBOOT**, while the new CircuitPlayground shows up as **CPLAYBOOT**, Trinket M0 will show up as **TRINKETBOOT**, and Gemma M0 will show up as **GEMMABOOT**.

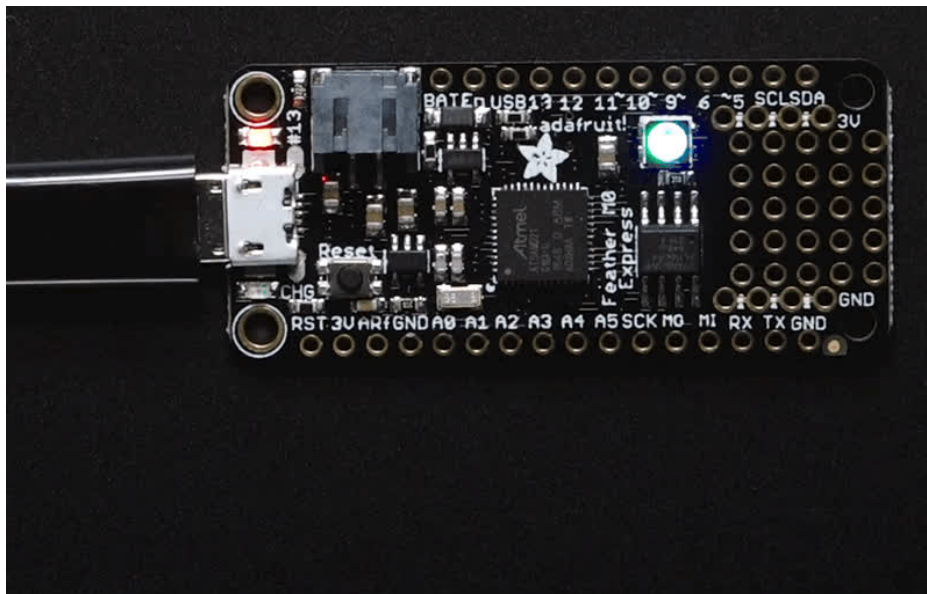
Furthermore, when the bootloader is active, it will change the color of one or more onboard neopixels to indicate the connection status, red for disconnected and green for connected. If the board is plugged in but still showing that its disconnected, try a different USB cable. Some cables only provide power with no communication.

For example, here is a Feather M0 Express running a colorful Neopixel swirl. When the reset button is double clicked (about half second between each click) the NeoPixel will stay green to let you know the bootloader is active. When the reset button is clicked once, the 'user program' (NeoPixel color swirl) restarts.

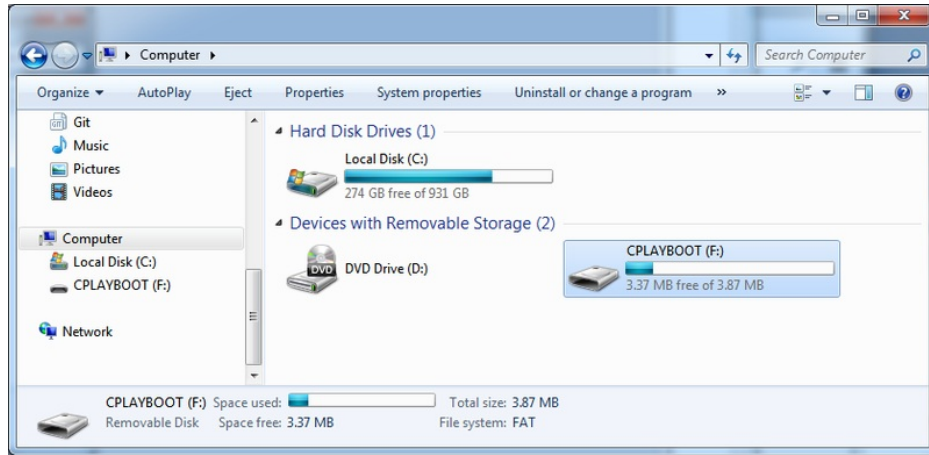


If the bootloader couldn't start, you will get a red NeoPixel LED.

That could mean that your USB cable is no good, it isn't connected to a computer, or maybe the drivers could not enumerate. Try a new USB cable first. Then try another port on your computer!

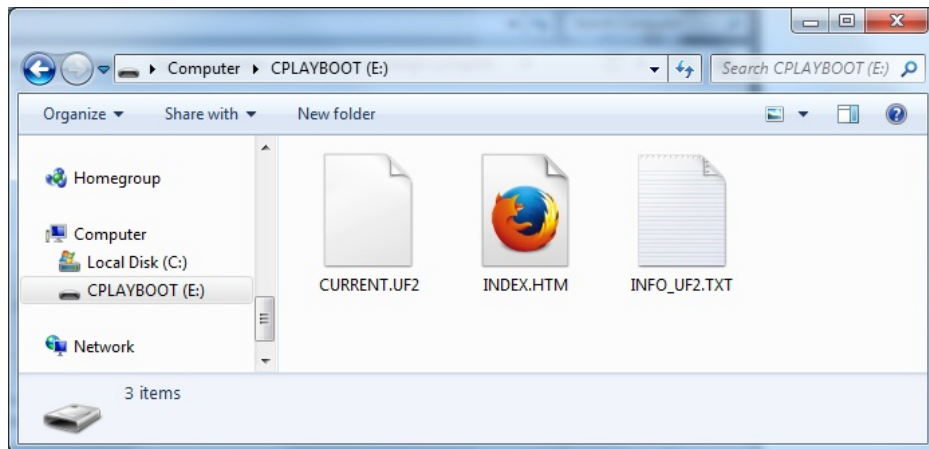


Once the bootloader is running, check your computer. You should see a USB Disk drive...



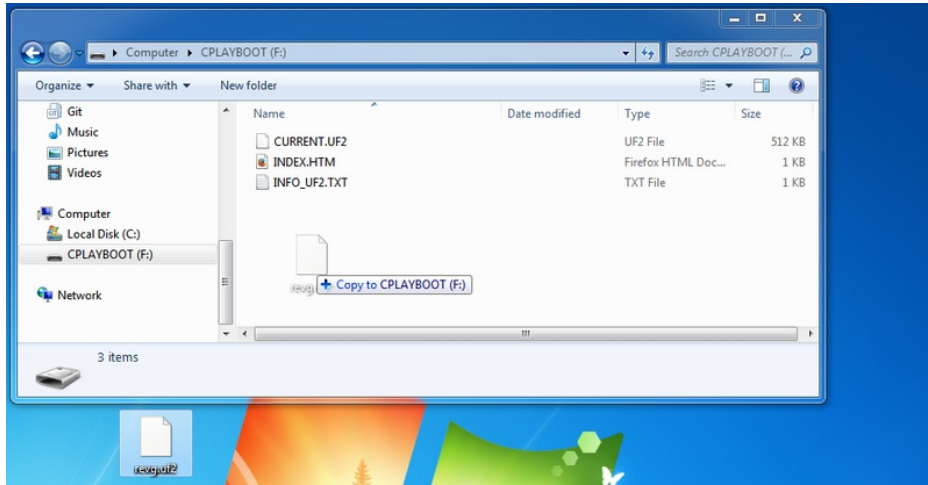
Once the bootloader is successfully connected you can open the drive and browse the virtual filesystem. This isn't the same filesystem as you use with CircuitPython or Arduino. It should have three files:

- **CURRENT.UF2** - The current contents of the microcontroller flash.
- **INDEX.HTM** - Links to Microsoft MakeCode.
- **INFO_UF2.TXT** - Includes bootloader version info. Please include it on bug reports.

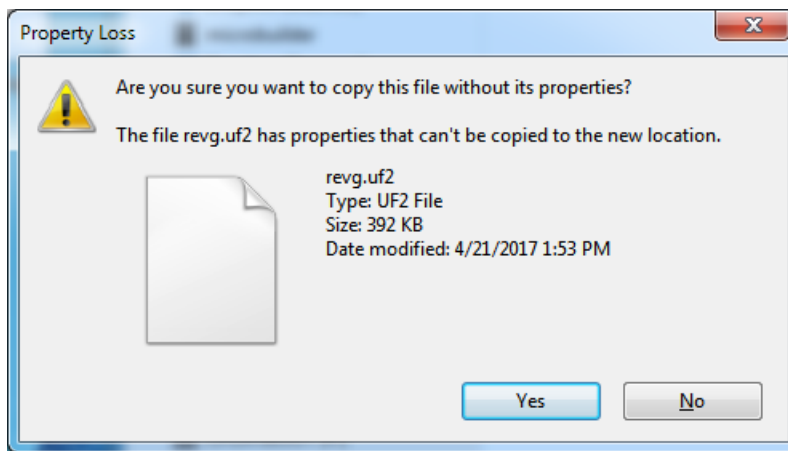


Using the Mass Storage Bootloader

To flash something new, simply drag any UF2 onto the drive. After the file is finished copying, the bootloader will automatically restart. This usually causes a warning about an unsafe eject of the drive. However, its not a problem. The bootloader knows when everything is copied successfully.



You may get an alert from the OS that the file is being copied without its properties. You can just click **Yes**



You may also get a complaint that the drive was ejected without warning. Don't worry about this. The drive only ejects once the bootloader has verified and completed the process of writing the new code

Using the BOSSA Bootloader

As mentioned before, the bootloader is also compatible with BOSSA, which is the standard method of updating boards when in the Arduino IDE. It is a command-line tool that can be used in any operating system. We won't cover the full use of the **bossac** tool, suffice to say it can do quite a bit! More information is available at [ShumaTech \(https://adafru.it/vQa\)](https://adafru.it/vQa).

Windows 7 Drivers

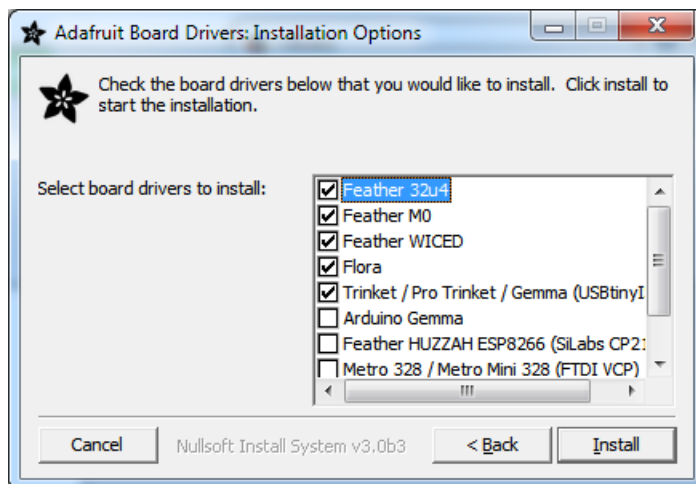
If you are running Windows 7 (or, goodness, something earlier?) You will need a Serial Port driver file. Windows 10 users do not need this so skip this step.

You can download our full driver package here:

<https://adafru.it/AB0>

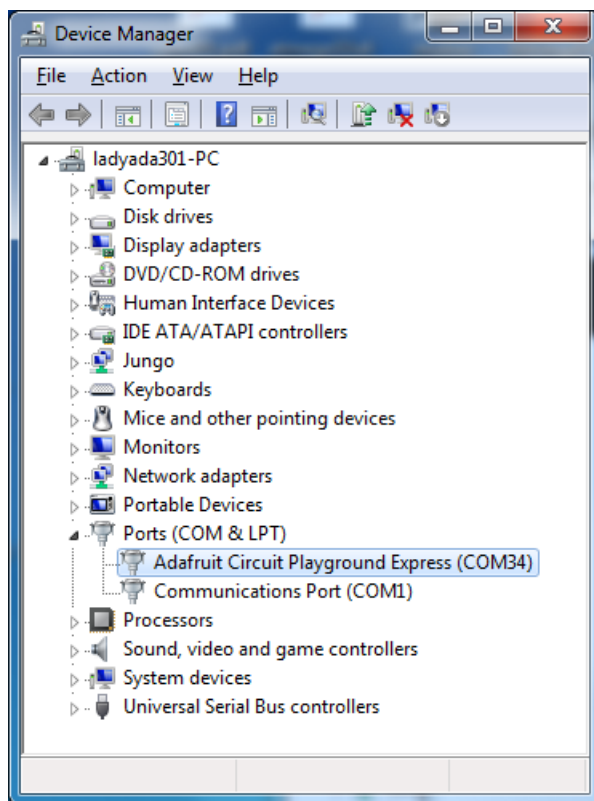
<https://adafru.it/AB0>

Download and run the installer. We recommend just selecting all the serial port drivers available (no harm to do so) and installing them.

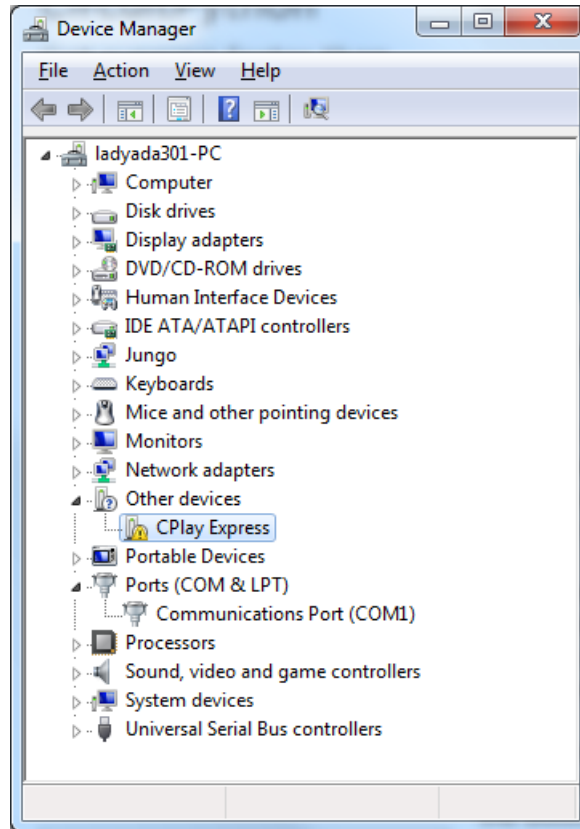


Verifying Serial Port in Device Manager

If you're running Windows, its a good idea to verify the device showed up. Open your Device Manager from the control panel and look under **Ports (COM & LPT)** for a device called **Feather M0** or **Circuit Playground** or whatever!



If you see something like this, it means you did not install the drivers. Go back and try again, then remove and re-plug the USB cable for your board



Running bossac on the command line

If you are using the Arduino IDE, this step is not required. But sometimes you want to read/write custom binary files, say for loading CircuitPython or your own code. We recommend using bossac v 1.7.0 (or greater), which has been tested. [The Arduino branch is most recommended \(https://adafru.it/vQb\)](https://adafru.it/vQb).

You can download the latest builds [here](https://adafru.it/s1B). (<https://adafru.it/s1B>) The `mingw32` version is for Windows, `apple-darwin` for Mac OSX and various `linux` options for Linux. Once downloaded, extract the files from the zip and open the command line to the directory with `bossac`.

With bossac versions 1.9 or later, you must use the `--offset` parameter on the command line, and it must have the correct value for your board.

With bossac version 1.9 or later, you must give an `--offset` parameter on the command line to specify where to start writing the firmware in flash memory. This parameter was added in bossac 1.8.0 with a default of `0x2000`, but starting in 1.9, the default offset was changed to `0x0000`, which is not what you want in most cases. If you omit the argument for bossac 1.9 or later, you will probably see a "Verify Failed" error from bossac. Remember to change the option for `-p` or `--port` to match the port on your Mac.

Replace the filename below with the name of your downloaded `.bin`: it will vary based on your board!

Using bossac Versions 1.7.0, 1.8

There is no `--offset` parameter available. Use a command line like this:

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R adafruit-circuitpython-boardname-version.bin
```

For example,

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R adafruit-circuitpython-feather_m0_express-3.0.0.bin
```

Using bossac Versions 1.9 or Later

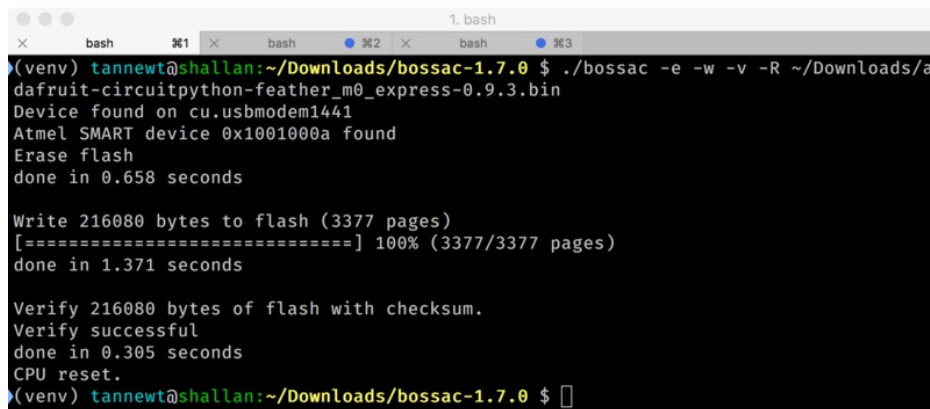
For M0 boards, which have an 8kB bootloader, you must specify `-offset=0x2000`, for example:

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R --offset=0x2000 adafruit-circuitpython-feather_m0_express-3.0.0.bin
```

For M4 boards, which have a 16kB bootloader, you must specify `-offset=0x4000`, for example:

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R --offset=0x4000 adafruit-circuitpython-feather_m4_express-3.0.0.bin
```

This will **e**rase the chip, **w**rite the given file, **v**erify the write and **R**eset the board. On Linux or MacOS you may need to run this command with `sudo ./bossac ...`, or add yourself to the `dialout` group first.



```
(venv) tannewt@shallan:~/Downloads/bossac-1.7.0 $ ./bossac -e -w -v -R ~/Downloads/a
dafruit-circuitpython-feather_m0_express-0.9.3.bin
Device found on cu.usbmodem1441
Atmel SMART device 0x1001000a found
Erase flash
done in 0.658 seconds

Write 216080 bytes to flash (3377 pages)
[=====] 100% (3377/3377 pages)
done in 1.371 seconds

Verify 216080 bytes of flash with checksum.
Verify successful
done in 0.305 seconds
CPU reset.
(venv) tannewt@shallan:~/Downloads/bossac-1.7.0 $
```

Updating the bootloader

The UF2 bootloader is relatively new and while we've done a ton of testing, it may contain bugs. Usually these bugs effect reliability rather than fully preventing the bootloader from working. If the bootloader is flaky then you can try updating the bootloader itself to potentially improve reliability.

If you're using MakeCode on a Mac, you need to make sure to upload the bootloader to avoid a serious problem with newer versions of MacOS. See instructions and more details [here \(https://adafru.it/ECU\)](https://adafru.it/ECU).

In general, you shouldn't have to update the bootloader! If you do think you're having bootloader related issues, please post in the forums or discord.

Updating the bootloader is as easy as flashing CircuitPython, Arduino or MakeCode. Simply enter the bootloader as above and then drag the `update bootloader uf2` file below. This uf2 contains a program which will unlock the bootloader section, update the bootloader, and re-lock it. It will overwrite your existing code such as CircuitPython or Arduino so make sure everything is backed up!

After the file is copied over, the bootloader will be updated and appear again. The `INFO_UF2.TXT` file should show the newer version number inside.

For example:

UF2 Bootloader v2.0.0-adafruit.5 SFHWRO

Model: Metro M0

Board-ID: SAMD21G18A-Metro-v0

Lastly, reload your code from Arduino or MakeCode or flash the [latest CircuitPython core \(https://adafru.it/Em8\)](https://adafru.it/Em8).

Below are the latest updaters for various boards. The latest versions can always be found [here \(https://adafru.it/Bmg\)](https://adafru.it/Bmg). Look for the `update-bootloader...` files, not the `bootloader...` files.

<https://adafru.it/ECV>

<https://adafru.it/ECV>

<https://adafru.it/ECW>

<https://adafru.it/ECW>

<https://adafru.it/ECY>

<https://adafru.it/ECY>

<https://adafru.it/ED0>

<https://adafru.it/ED0>

<https://adafru.it/ED3>

<https://adafru.it/ED3>

<https://adafru.it/ED6>

<https://adafru.it/ED6>

<https://adafru.it/ED8>

<https://adafru.it/ED8>

<https://adafru.it/Bmg>

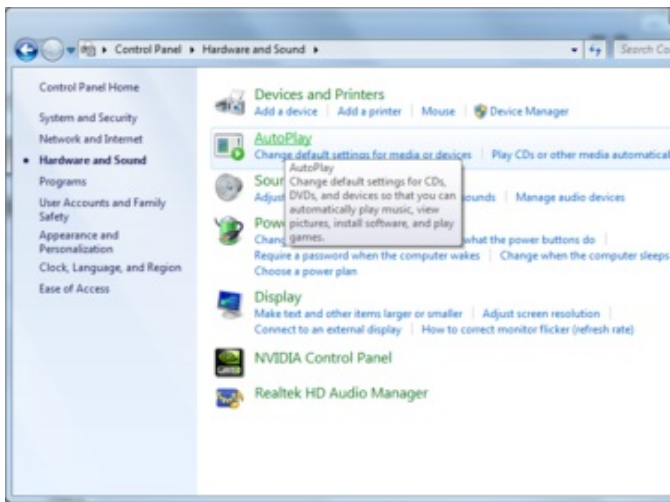
<https://adafru.it/Bmg>

Getting Rid of Windows Pop-ups

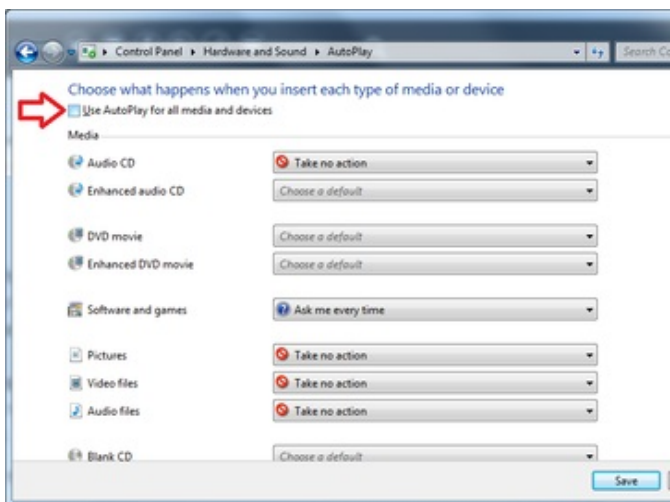
If you do a *lot* of development on Windows with the UF2 bootloader, you may get annoyed by the constant "Hey you inserted a drive what do you want to do" pop-ups.



Go to the Control Panel. Click on the **Hardware and Sound** header



Click on the **AutoPlay** header



Uncheck the box at the top, labeled **Use AutoPlay for all devices**

Making your own UF2

Making your own UF2 is easy! All you need is a .bin file of a program you wish to flash and [the Python conversion script \(https://adafru.it/vZb\)](https://adafru.it/vZb). Make sure that your program was compiled to start at 0x2000 (8k) for M0 boards or 0x4000 (16kB) for M4 boards. The bootloader takes up the first 8kB (M0) or 16kB (M4). CircuitPython's [linker script \(https://adafru.it/CXh\)](https://adafru.it/CXh) is an example on how to do that.

Once you have a .bin file, you simply need to run the Python conversion script over it. Here is an example from the directory with **uf2conv.py**. This command will produce a **firmware.uf2** file in the same directory as the source **firmware.bin**. The uf2 can then be flashed in the same way as above.

```
# For programs with 0x2000 offset (default)
uf2conv.py -c -o build-circuitplayground_express/firmware.uf2 build-circuitplayground_express/firmware.bi

# For programs needing 0x4000 offset (M4 boards)
uf2conv.py -c -b 0x4000 -o build-metro_m4_express/firmware.uf2 build-metro_M4_express/firmware.bin
```

Installing the bootloader on a fresh/bricked board

If you somehow damaged your bootloader or maybe you have a new board, you can use a JLink to re-install it. [Here's a short writeup by turbinenreiter on how to do it for the Feather M4 \(but adaptable to other boards\) \(https://adafru.it/ven\)](https://adafru.it/ven)

Downloads

<https://adafru.it/B48>

<https://adafru.it/B48>

Datasheets

- [ATSAMD21 Datasheet \(https://adafru.it/kUf\)](https://adafru.it/kUf) (the main chip on the Feather M0)
- [Fritzing object in the Adafruit Fritzing Library \(https://adafru.it/aP3\)](https://adafru.it/aP3)
- [EagleCAD PCB files in GitHub \(https://adafru.it/vfS\)](https://adafru.it/vfS)

<https://adafru.it/z3F>

<https://adafru.it/z3F>

Note AREF in the diagram should be marked PA03 not PA02

Firmware

'Classic' Feather M0 Bootloader - You'll need to program it in using an ST-Link, JLink or other SWD-capable programmer. [HEX available in the github repo \(https://adafru.it/kFh\)](https://adafru.it/kFh)

Schematic & Fabrication Print

